

# **Data Structures**

## **Class Notes**

**© Copyright 2000 2001 2002 2003 2004 2005  
2006 2007 2008 2009 2010 2011 2012 2013  
2014, 2015, 2016**

**by David Burris, Ph.D., CCP, CSP**

**Permission to use this material for personnel educational purposes  
by students currently registered in my classes is granted. All other  
rights reserved.**

# Contents

1	<a href="#">Basic Notation Using a Deck of Cards</a>	4
2	<a href="#">List Structure Classification</a>	9
3	<a href="#">Sequentially Allocation Basics, Stack, and Queue</a>	11
4	<a href="#">Bounded Linearly Allocated Circular Queue</a>	14
5	<a href="#">Bounded Linearly Allocated Circular Deque</a>	15
	Implementation Examples	16
	Special Case of Two Stacks	17
6	<a href="#">Multiple Sequentially Allocated Stacks</a>	18
	Random Insert, Random Delete, Insert Sorted Contiguous Allocation	25
7	<a href="#">Stack implemented as a link allocated linear list</a>	28
8	<a href="#">Dynamic Storage Allocation Algorithm</a>	29
	Implementation Examples	30
9	<a href="#">Link allocated Queue</a>	32
10	<a href="#">Spock, Mission Impossible</a>	36
11	<a href="#">Link Sort into groups of all records with the same Key</a>	37
12	<a href="#">Linked list operations, insert sorted (lexicographic), random delete</a>	47
13	<a href="#">Efficient Bi-Directional Traversal of a Singly Linked List</a>	49
14	<a href="#">Topological Sort</a>	52
15	<a href="#">Doubly Linked List</a>	63
16	<a href="#">Link Allocated Circular Lists</a>	65
17	<a href="#">Polynomial Arithmetic</a>	68
18	<a href="#">Subscripting Schemes</a>	71
19	<a href="#">Introduction to Trees and Binary Trees</a>	75
20	<a href="#">Threaded Trees</a>	83
22	<a href="#">Binary (Alphabetic) Tree Search, Insertion, and Deletion</a>	90
23	<a href="#">Forest / M-Ary Trees</a>	95
24	<a href="#">Artificial Intelligence, Game of 16 &amp; 8</a>	96
25	<a href="#">Post Order by Degrees</a>	102
26	<a href="#">Determining Equivalence</a> Sets	103
27	<a href="#">Binary Search on Linearly Allocated List</a>	105
28	<a href="#">Ill Spelled Names</a>	108
29	Ordering of list: frequency of use, self ordering	110
21	<a href="#">B-Tree</a>	111
30	<a href="#">Sorting: Simple Bubble, True Bubble, Selection, Shell, Quicksort, Heap, Hash, Radix, Straight Insertion, Link Insertion</a>	123
31	<a href="#">Significant factors in sorting</a>	136
32	<a href="#">Hashing</a>	138
33	<a href="#">File Organization and Access Methods</a>	166
	Priority Queue	171

		Ada package Text_IO	187	
		Multi-list and Inverted List		
		Activity Oriented List		
		Sorting Sequential Files in Place		
		Warshall's and Wirth-Weber Relations	193	

## Basic Notation Using a Deck of Cards

Partial hand  
of cards.

Face Down  
3 of Hearts

Hole  
Card

Face up  
8 of Spades

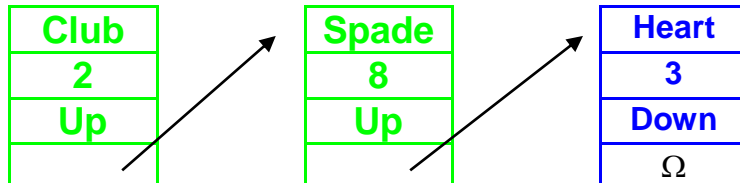
Face Up  
2 of Clubs

### Card Format

- 1) **Suit:** (clubs, diamonds, hearts, or spades).
- 2) **Rank:** 1 through king.
- 3) **Facing:** up or down.
- 4) **Next:** point to next card in list.

Cards on the left dealt as 3 of hearts down, followed by the 8 of spades up, and finally the 2 of clubs up.

Hand



Ada	"C++"
<pre> type Suit is ( Club, Diamond,                 Heart, Spade ); type Facing is ( Up, Down );  type Card; type CardPt is access Card;  type Card is tagged record     cardSuit: Suit;     cardRank: Integer range 1..13;     cardFacing: Facing;     next: CardPt; end;</pre>	<pre> enum Suit {Club, Diamond, Heart,             Spade }; enum Facing {Up, Down };  class Card{ //or struct Card C/C++     Suit cardSuit;     int cardRank;     Facing cardFacing;     Card *next     // methods follow };</pre>

## To deal a single hand pointed to by Hand.

We assume a function, “Avail,” which allocates a node (Card of storage) each time it is invoked. This algorithm assigns values to a single card and adds the card to the list pointed to by Hand. We assume the list pointed to by Hand is initially empty, i.e.,  $\text{Hand} \leftarrow \Omega$ . Pt and Hand are of type CardPoint, i.e., they are pointers to nodes of storage of type Card.

- 1) [allocate card and set to proper values]  
**Pt <= Avail;** -- Implies to allocate a node of storage and  
-- set Pt pointing to it. *Should check for overflow!*
- 2) **Pt.cardSuit ← x;** -- Set Suit field of node pointed to by Pt to the  
-- value of the variable x.
- 3) **Pt.cardRank ← x;** -- Set Rank field of card pointed to by PT  
-- to desired value.
- 4) **Pt.cardFacing ← x;** -- Indicate if card is face up or down.
- 5) [Add card to list]  
**Pt.next ← Hand;**  
**Hand ← Pt.**

**To count the cards currently in the hand pointed to by Hand.**

- ```

1)  Knt  $\leftarrow$  0;
2)  Pt  $\leftarrow$  Hand;
3)  While Pt  $\neq \Omega$  loop
        Knt  $\leftarrow$  Knt + 1;
        Pt  $\leftarrow$  Pt.next;
    End loop;
4)  Print the value of Knt.

```

**Multiple hands (to represent 5 players) could be expressed by**

**Hands: array(1..5) of CardPt;**

-- Cards.adb. This program deals four cards then counts the  
-- cards in the linked list.

with Ada.Text\_IO; use Ada.Text\_IO;  
procedure Cards is

package IntIO is new Ada.Text\_IO.Integer\_IO(integer);  
use IntIO;

type Card;  
type CardLink is access Card;

type Card is record

Suit: Character;

Rank: integer; -- 1=> ace, 10 => 10, 11 => jack, 12 => queen 13 => king.

Facing: character; -- U => up, D => down.

Next: CardLink;

end record;

Hand, Pt: CardLink; -- initially set null at compile time.

Knt: integer;

begin

for I in 1..4 loop -- Deal 4 cards.

Pt := new Card;

put("Enter the suit (C, D, H, S): "); get( Pt.Suit );

put("Enter the Rank (1 thru 13): "); get( Pt.Rank );

put("Enter facing (U => up, D => down): "); get( Pt.Facing );

-- add to list

Pt.Next := Hand;

Hand := Pt;

end loop;

```

-- Count the number of cards in the hand.
Knt := 0;
Pt := Hand;

while Pt /= null loop -- print hand
    put( Pt.Suit ); put(" "); put( Pt.Rank ); put(" ");
    put( Pt.Facing ); new_line;

    Knt := Knt + 1;
    Pt := Pt.Next; -- Point to the next card.
end loop;

put("Total cards = "); put(Knt);
end;

```

```

C:\>Cards
Enter the suit (C, D, H, S): C
Enter the Rank (1 thru 13): 2
Enter facing (U => up, D => down): D
Enter the suit (C, D, H, S): D
Enter the Rank (1 thru 13): 1
Enter facing (U => up, D => down): U
Enter the suit (C, D, H, S): S
Enter the Rank (1 thru 13): 3
Enter facing (U => up, D => down): U
Enter the suit (C, D, H, S): H
Enter the Rank (1 thru 13): 3
Enter facing (U => up, D => down): U
C                2 D
D                1 U
S                3 U
H                3 u
Total cards =                4

```

```

-- Cards2.adb, data in Cards2In.txt
with Ada.Text_IO; use Ada.Text_IO;
procedure Cards2 is

    type SuitType is ( Club, Diamond, Heart, Spade );
    type FacingType is ( Up, Down );

    package IntIO is new Ada.Text_IO.Integer_IO(integer);
    use IntIO;

    package Suit_IO is new Ada.Text_IO Enumeration_IO(SuitType);
    use Suit_IO;

    package Facing_IO is new Ada.Text_IO Enumeration_IO(FacingType);
    use Facing_IO;

    type Card;
    type CardLink is access Card;

    type Card is record
        Suit: SuitType;
        Rank: integer; -- 1=> ace, 10 => 10, 11 => jack, 12 => queen 13 => king.
        Facing: FacingType;
        Next: CardLink;
    end record;

    Hand, Pt: CardLink;
    Knt: integer;
begin

    for I in 1..4 loop -- Deal 4 cards.
        Pt := new Card;
        put("Enter the suit (Club, Diamond, Heart, Spade): "); get( Pt.Suit );
        put("Enter the Rank (1 thru 13): "); get( Pt.Rank );
        put("Enter facing (Up, Down): "); get( Pt.Facing );

        -- add to list
        Pt.Next := Hand;
        Hand := Pt;
    end loop;

```



```

-- Count the number of cards in the hand.
Knt := 0;
Pt := Hand;

while Pt /= null loop -- print hand
    put( Pt.Suit ); put(" "); put( Pt.Rank ); put(" ");
    put( Pt.Facing ); new_line;

    Knt := Knt + 1;
    Pt := Pt.Next; -- Point to the next card.
end loop;

put("Total cards = "); put(Knt);
end;

```

```

// Use with PlayCards.java
public class Card {
    public enum Suit { Spade, Heart, Diamond, Club, None };
    public enum Facing { Up, Down, Side };

    private Suit suit;
    private int rank;
    private Facing facing;
    private Card nextCard;

    Card ( ) { // default constructor
        suit = Suit.None; rank = 0; facing = Facing.Side; nextCard = null;
    }

    Card(Suit suitIn, int rankIn, Facing facingIn, Card cardIn) {
        suit = suitIn; rank = rankIn; facing = facingIn; nextCard = cardIn;
    }

    // Set and Access functions.
    public void setSuit( Suit suitIn ) { suit = suitIn; }
    public Suit getSuit( ) { return suit; }

    public void setRank( int rankIn ) { rank = rankIn; }
    public int getRank( ) { return rank; }

    public void setFacing( Facing facingIn ) { facing = facingIn; }
    public Facing getFacing( ) { return facing; }

    public void setNextCard( Card cardIn ) { nextCard = cardIn; }

    public Card getNextCard ( ) { return nextCard; }
} // End class Card

//Uses class Card.java
import java.io.*;
public class PlayCards {
    public static void main( String args[ ] ) {
        Card hand, aCard;

        aCard = new Card( );
        aCard.setSuit( Card.Suit.Heart );
        aCard.setRank( 3);
        aCard.setFacing( Card.Facing.Down );
        aCard.setNextCard( null );
        hand = aCard;

        hand = new Card( Card.Suit.Spade, 8, Card.Facing.Up, hand );

        hand = new Card( Card.Suit.Club, 2, Card.Facing.Up, hand );

        aCard = hand;
        while( aCard != null) {
            System.out.println( aCard.getSuit() );
            System.out.println( aCard.getRank() );
            System.out.println( aCard.getFacing() );
            aCard = aCard.getNextCard();
            System.out.println(" ");
        }
    }
}

```

# List Structure Classification

We tend to classify "list" structures by:

## A) the method for storage allocation

- a) **static/sequential** (we know exact number of items or there is a reasonable upper bound, contiguous allocation)
- b) **dynamic** (linked list, no prior knowledge on the number of items)

## B) the operations we perform on them,

- i) randomly access the  $k^{\text{th}}$  node – *Prefer sequential*
- ii) randomly insert just before or after the  $k^{\text{th}}$  node – *Prefer Dynamic (linked list)*
- iii) randomly delete the  $k^{\text{th}}$  node – *prefer dynamic*
- iv) combine two or more list into one list - *Dynamic*
- v) split a list into two or more list *either, faster if sequential*
- vi) make a copy of a list – *Either, depends on the program (counters, etc)*
- vii) determine the number of nodes in a list – *Sequential*
- viii) sort the list – *depends on application, faster with sequential*
- ix) search for a node containing a specific value - *Dynamic*

## Basic Structures

- 1) **Stack** (operations push and pop) (LIFO: last-in-first-out)
- 2) **Queue** (operations insert rear, remove from front) (FIFO: first-in-first-out)
- 3) **Deque** (insert and remove from both ends)

# Sequential Allocation Basics

Assume we wish to store information in sequentially (contiguously) allocated memory. This is frequently referred to as an array. In general, the elements of an array are of a structured type requiring “C” basic units of storage on a computer. The following assumes a character oriented memory structure with 2 bytes per integer and one byte per character.

```
type MyType is record
    F1: array(1..3) of character;
    F2: integer; --assume 2
                -- bytes/integer
end record;
```

```
X: array(0..99) of MyType;
```

Assume the array X starts at base address 100 in real memory. Then the location of a random element “J” may be calculated as:

$$\text{Loc}[X(J)] = \text{Base} + \text{Offset} \\ = \text{Base} + C * J$$

for zero base indexing where C is the number of units of storage in a logical entry. For the data structure X above,

$$\text{Loc}[X(J)] = \text{Base} + 5 * J, \text{ as } C = 5.$$

The location of the next node is:

$$\text{Loc}[X(J+1)] = \text{Loc}[X(J)] + C.$$

\*\* Assume the following declaration:

```
Y: Array(1..100) of MyType;
```

A random location would be adding or subtracting a constant to make the start address relative to zero (translate the axis):

File

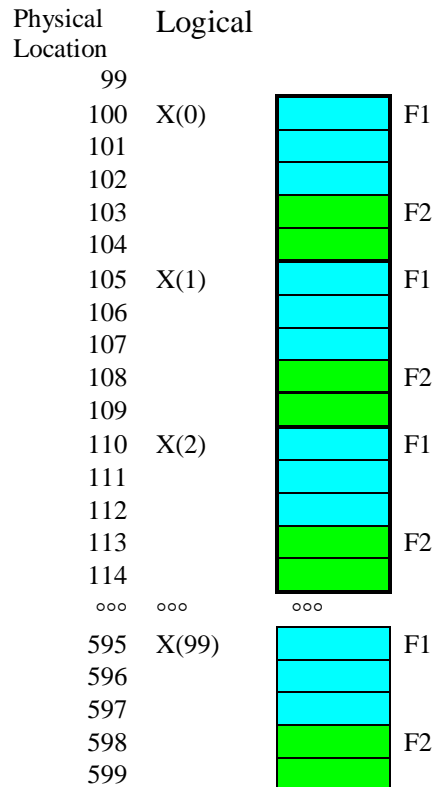
$$\text{Loc}[X(J)] = \text{Base} + C * (J-1).$$

$$\text{Or } \text{Loc}[X(J)] = \text{Base} + C * J$$

$$\text{for Base} = \text{Start} - C = 100 - 5 = 95.$$

Usage: Calculating subscripts in assembly language, and used by language translator writers. In real time environments it may be faster to calculate subscripts directly than utilize the more convenient subscripting ability of a high level language.

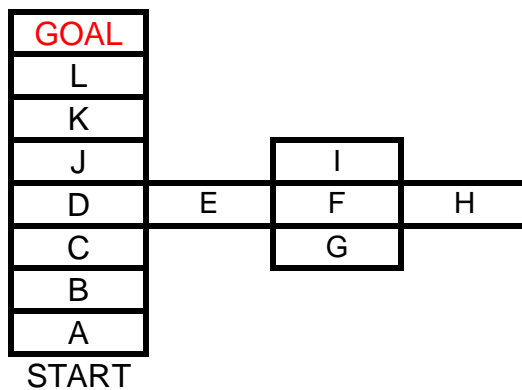
Array of 3 character stock identifiers and number of items in stock.



# Stacks

Stacks are the most widely used data structure in computing. They are frequently referred to as LIFO list, as the last item placed in the stack is the first to be removed. Typically stacks support two operations, “push” and “pop.” Other stack methods such as “how many items are currently in the stack,” “is the stack empty,” etcetera are convenient.

Consider the problem of assembling a space station in space. To reduce risk, it has been decided astronauts should be tethered to the incomplete station or ship as much as possible. Assume we have been contracted to build Robby Robot. Robby is supposed to find tools and parts required for the assembly process and bring them to the astronaut. This was an actual NASA project eventually dropped for weight, power, and cost restrictions. The actual search would have been in 3-D. A partial solution in 2-D follows incorporating a stack. If the goal was not in site, the robot would move to the next grid coordinate counter clockwise (starting below the current location) from its current location as long as it had not previously visited that location after pushing the location into the stack. Once the item was found, the robot returned to the astronaut by popping the stack.



The initial trip would be A, B, C, D, E, F, G, H, I, J, K, L, GOAL with the stack appearing as follows where position with lines through them(E, F, G, H, and I) were popped out of the stack during the search:



The “return trip” would be accomplished by simply popping the remaining grid coordinates from the stack: GOAL, L, K, J, D, C, B, A, START. This represents the most direct path back to the astronaut.

## Sequentially Allocated Stack

Assume we implement the stack as an array of 1 to Max items. The variable “top” points to the top element of the stack. Top = 0 will be the boundary condition to indicate an empty stack. For convenience, we assume a high level language so that C = 1.

```
max: integer := 10;
type MyType is record
    F1: array(1..max) of character;
    F2: integer;
End record;

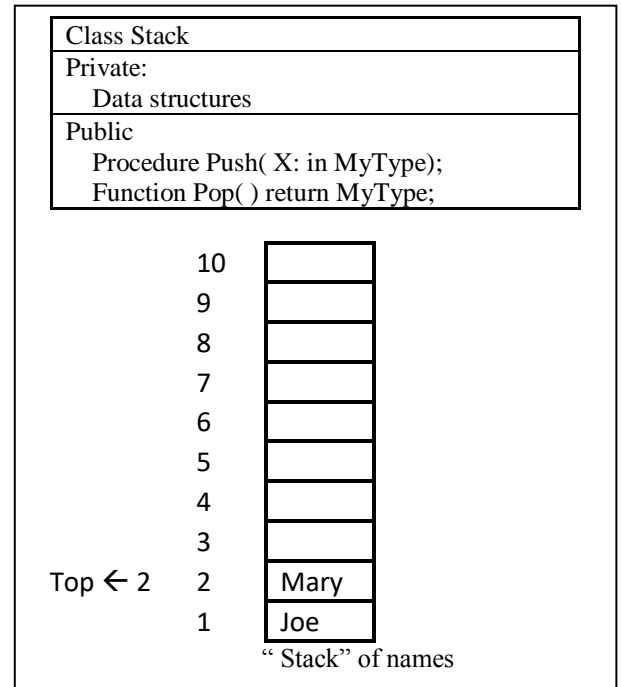
stack: array(1..10) of MyType;
top: integer range 0..10 := 0; -- Set empty.
```

### Method Push

```
Procedure Push( X: in MyType) is
begin
    If( top < max) then
        -- Insert item X into stack.
        top ← top + 1; -- top ← top + c;
        Stack( top ) ← X;
    else
        report overflow; -- This is normally an error condition.
    end if;
end Push;
```

### Method POP:

```
function Pop( ) return MyType is
begin
    If( top > 0 ) then
        top ← top – 1; -- top ← top – c;
        return stack( top + 1); -- Pop the stack. => return stack( top + c);
    else
        report underflow; -- This is normally a desired condition.
    End if;
end Pop;
```



## Sequentially Allocated Queue

Assume we implement the queue as an array of Max items. The variable “front” points to the next available element in the queue. “Rear” points to the last item in the queue. We assume all insertions at the rear and all removals from the front. Front = rear will be the boundary condition to indicate an empty queue. For initialization, we set front = rear = 0. We implement the queue under the assumption that the queue empties on a regular basis, i.e., the front = rear when the last item is removed from the queue. When the queue becomes empty, we reset front = rear = 0 to reclaim previously used space (prevent the queue from constantly growing in the same direction). The following queue is only practical if all data in the queue can be processed on a regular basis. For convenience, we assume a high level language so that C = 1 (the distance from one queue element to the next). Through out this algorithm the pointer front is usually one less than the actual location of the first element of the queue. The pointer rear actually points to the last item in the queue if the queue is not empty.

```
max: integer := 6;
type MyType is record
    F1: array(1..10) of character;
    F2: integer;
End record;

Queue: array(1..6) of MyType;
front, rear: integer range 0..6 := 0;
```

|                                   |
|-----------------------------------|
| Class Queue                       |
| Private:                          |
| Data structures                   |
| Public                            |
| Procedure Insert( X: in MyType);  |
| Function Remove( ) return MyType; |

### Insert in rear of the queue:

```
If ( rear < max ) then
    rear ← rear + 1; -- rear ← rear + c;
    Queue( rear ) ← X; -- Insert X in rear of queue.
else
    Report overflow; -- Normally an error condition.
end if;
```

### Remove from the front of the queue:

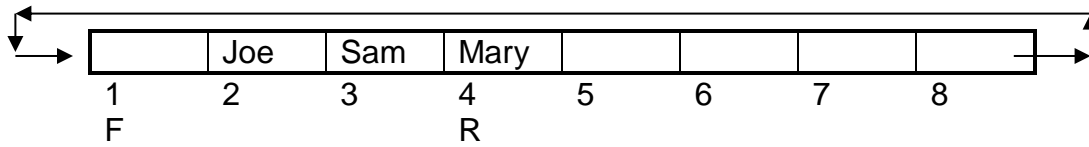
```
If( front /= rear) then
    front ← front + 1; -- front ← front + c;
    X ← Queue( front ); -- Set X to item at front of queue removing it.
    If( front = rear ) then front ← rear ← 0; -- Reset, queue is empty.
else
    report underflow; -- Normally a desired condition.
end if;
```

Queue:

| 0     | 1   | 2   | 3     | 4 | 5 | Max = 6 |
|-------|-----|-----|-------|---|---|---------|
|       | Joe | Tom | Sue   |   |   |         |
| F = 0 |     |     | R = 3 |   |   |         |

This implementation of a queue is most useful when the queue empties on a periodic basis, e.g., an earth satellite dumping over the home antenna, polling gas stations, ATM,s etcetera for all transactions. Location 0 is never physically used.

### Bounded Linearly Allocated Circular Queue



Assume a bounded circular queue of  $M$  items. In the above diagram  $M = 8$ . We insert at the rear ( $R$ ) of the queue and remove from the front ( $F$ ) of the queue. We assume the boundary condition that  $F \leftarrow R \leftarrow 1$  initially. We wish to treat the queue as empty thereafter anytime  $F = R$ .

Assume the queue is named "X" and that "Y" represents a unit of data.

**Operation InsertRear** [  $X \leq Y$ , insert Y into the queue X (rear, on the right)]

- 1) If  $R = M$  then  
     $R \leftarrow 1$   
Else  
     $R \leftarrow R + 1$   
End If;
- 2) If  $R = F$  then  
    Report overflow and stop  
Else  
     $X[R] \leftarrow Y$   
End If;

**Operation DeleteFront** [  $Y \leftarrow X$ , assign Y the value at the front of queue X (left)]

- 1) If  $R = F$  then  
    Report underflow and stop  
Else  
    If  $F = M$  then  
         $F \leftarrow 1$   
    Else  
         $F \leftarrow F + 1$   
    End If;  
     $Y \leftarrow X[F]$   
End If;

Note only  $M-1$  units of the queue can be used due to the boundary condition for underflow and overflow. All  $M$  units of memory could be used if a separate counter is used to keep track of the number of queue units actually in use. The tradeoff would be the space for the counter and overhead to maintain it on each queue operation.



# Make the queue into a *deque*.

**Operation InsertFront** [insert Y into the deque X as a new item on the left (front)]

- 1)     $LL \leftarrow F$   
      If  $F = 1$  then  
           $F \leftarrow M$   
      Else  
           $F \leftarrow F - 1$   
      End If;
- 2)    If  $F = R$  then  
          report overflow and stop  
      Else  
           $X[LL] \leftarrow Y$   
      End If;

**Operation DeleteRight** [Remove the next item on the right (rear)]

- 1)    If  $F = R$  then  
          Report underflow and stop  
      Else  
           $Y \leftarrow X[R]$   
      End if;
- 2)    If  $R = 1$  then  
           $R \leftarrow M$   
      Else  
           $R \leftarrow R - 1$   
      End If;

## Implementation Examples

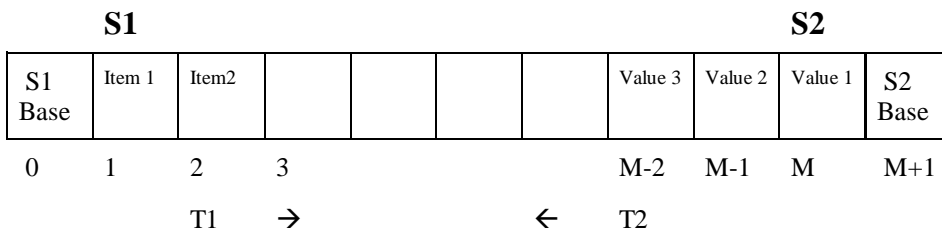
|  | Problem                                                   | Document                 | Page    |  |
|--|-----------------------------------------------------------|--------------------------|---------|--|
|  | Simple stack - dostack                                    | Data Structures Programs | 51      |  |
|  | Object Oriented Stack                                     | Data Structures Programs | 52      |  |
|  | Ada Generics                                              | Data Structures Programs | 53      |  |
|  | Generic Stack                                             | Data Structures Programs | 54      |  |
|  | Generic Circular Queue - SMailbox                         | Data Structures Programs | 55      |  |
|  | Dynamic storage allocation in system stack – box1         | Data Structures Programs | 57      |  |
|  | More sophisticated object oriented C++ stack – stack0.cpp | Data Structures Programs | 69      |  |
|  | C++ Template Stacks                                       | Data Structures Programs | 70 - 75 |  |
|  | Ada shot records and I/O routines                         | Data Structures Programs | 58      |  |
|  | Passing I/O routines - GIOEX                              | Data Structures Programs | 76 - 78 |  |
|  | Syntax for generic I/O                                    | Data Structures          |         |  |
|  |                                                           |                          |         |  |
|  |                                                           |                          |         |  |

Examples from Data Structures Programming.

## Special Case Involving Two Stacks

Consider the problem of two earth satellites in the same orbit on opposite sides of the earth and a single receiving tower. Communications is line of sight with the tower. As one satellite comes within range of the tower it starts broadcasting information stored in a stack reducing its size. At the same time the second satellite loses communications with the tower and must start storing data in its stack maximizing its size. We assume the stack of the broadcasting satellite will become empty prior to losing communications. The space required for the combined stacks  $\leq M$  at all times.

We wish to utilize two variable size stacks, S1 and S2, in the most efficient manner possible given M units of memory occupying locations 1 through M. Under normal circumstances the stacks grow and shrink dynamically such that at any given point in time either stack could consume considerably more than half of memory. However, it is anticipated that at no point in time will the combined size of the two stacks exceed main memory, i.e.,  $\text{size}(\text{stack1}) + \text{size}(\text{stack2}) \leq M$ . The following algorithm has been suggested where T1 and T2 are the tops of stacks S1 and S2 respectively. Note that stack S1 is growing from left to right and that stack S2 is growing from right to left. As a boundary condition, the initial value of  $T1 := 0$  and the initial value of  $T2 := M+1$ . Stack S1 is considered empty if  $T1 = 0$  and S2 is empty if  $T2 = M+1$ . Positions 0 and M+1 are not utilized. Assume the contiguous locations are referred to by "S."



|                                                                                                                                                   |                                                                                                                                                     |
|---------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| Insert in Stack S1:<br>$T1 := T1 + 1;$<br>If $T1 = T2$ then<br>Report overflow<br>else<br>$S[T1] := \text{item};$<br>End if                       | Insert in Stack S2:<br>$T2 := T2 - 1;$<br>If $T1 = T2$ then<br>Report overflow<br>else<br>$S[T2] := \text{item};$<br>End if                         |
| Delete from Stack S1:<br>1) If $T1 = 0$ then<br>Report underflow and<br>terminate the deletion;<br>End If;<br>2) $Y := S[T1];$<br>$T1 := T1 - 1;$ | Delete from Stack S2:<br>1) If $T2 = M+1$ then<br>Report underflow and terminate<br>the deletion;<br>End If;<br>2) $Y := S[T2];$<br>$T2 := T2 + 1;$ |

Unfortunately experience has shown that the user population frequently submits erroneous specifications. In particular, we distrust the claim that the combined size of the stacks will never exceed the size of main memory. Tests must be added to the above algorithms to check for overflow and underflow (you may not modify the basic strategy used by the algorithm). In the event that overflow should occur, reject the transaction and print an error message. The main procedure should continue to process transactions until all transactions have been processed. Use every unit of available memory prior to reporting an overflow condition.

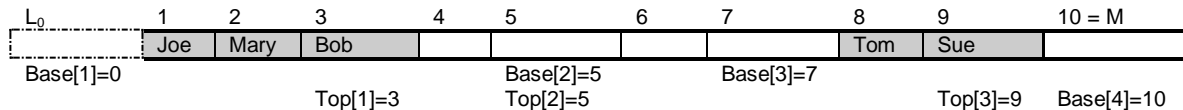
PS: A stack is frequently preferred to a queue for processing. A stack allows you to start with the most recent data which is potentially has the most immediate significance. A queue starts with the oldest data.

## Multiple Sequentially Allocated Lists Sharing Memory Locations

Assume the case where multiple sequentially allocated list must coexist in a restricted amount of memory. We assume that the list will vary in size dynamically. If one list overflows into space currently occupied by another list, we choose to rearrange the lists by moving their contents to accommodate the space requirements for the list that overflowed. For convenience, we will assume initially that all lists are stacks. We further assume for convenience that the stacks occupy memory locations 1 through M known as "StackSpace." All stacks are initially allocated equal amounts of space. The sized of each stack is then allowed to vary dynamically at run time. If Base[J] and Top[J] are used to track the space allocations for each stack, then initial space allocation is given by:

$$\text{Base}[J] = \text{Top}[J] = \text{floor}((J - 1) / N * M) + L_0$$

where N is the number of stacks,  $1 \leq J \leq N$ , and the stacks share the common memory area consisting of all locations L with  $L_0 < L \leq M$ . The "floor" operator means to truncate the fractional part of the number, e.g., floor(7.89699) is 7.



The diagram represents 3 stacks utilizing 10 memory locations 1 through 10. Note that a fourth "Base" has been allocated to allow for a consistent means to check for overflow in the insertion algorithm below. In general, for N stacks, we will use N + 1 bases.  $L_0$  is treated as a dummy location just prior to the useful memory area.

Comments are enclosed in { } in the following algorithms.

The basic stack operations are as follows:

### Insertion in stack K:

```

Top[K] := Top[K] + 1;
If Top[K] > Base[K + 1] then
    report overflow; {Algorithm Reallocate should be used to determine if
                    additional space can be made available by taking it from another stack.}
Else
    StackSpace[ Top[K] ] := Y; {Insert value of Y into the stack.}
End If;
```

### Deletion from stack K:

```
If Top[K] = Base[K] then
    report underflow;
Else
    Y := StackSpace[ Top[K] ]; {Remove top item in stack and assign to Y.}
    Top[K] := Top[K] - 1;
End If;
```

When overflow occurs in algorithm Insertion, one of three possibilities exist:

- A) Find the smallest J for which  $K < J \leq N$  and  $\text{Top}[J] < \text{Base}[J+1]$ . If any such J exist, move contents up one or more locations.
- B) If step "A" fails, find the largest J for which  $1 \leq J < K$  and  $\text{Top}[J] < \text{Base}[J+1]$ . If this condition exists, move items down one or more notches.
- C) We find  $\text{Top}[J] = \text{Base}[J+1]$  for all  $J \neq K$ . We are out of memory and no accommodation is possible for the overflow condition.

### Algorithm Reallocate (sequentially allocated tables).

We assume that overflow has occurred in one of the N stacks pointed to by  $\text{Top}[K]$  in the space allocation "StackSpace."  $\text{Top}[J]$  is the current top for stack "J."  $\text{Base}[J]$  represents the current base for each stack. This algorithm will reallocate available memory between stacks. EqualAllocate is the percentage of available memory that should be allocated equally between all stacks. GrowthAllocate is the percent of available memory that should be allocated to stacks based on their growth since the last time memory overflowed. For example, if EqualAllocate = 0.6, then 60% of free memory should be shared equally by all stacks on overflow and 40% should be used to reflect growth. Three temporary arrays OldTop, Growth, and NewBase are required. **They may actually occupy the same physical memory locations (a single physical array) if mapped in the following manner:  $\text{OldTop}[J] = \text{Growth}[J - 1] = \text{NewBase}[J]$  for  $1 \leq J \leq (N+1)$ .** OldTop[J] should initially be set to the initial value of the top of stack pointer for each stack when the main program is initiated. We assume the stacks occupy all memory locations L where  $1 \leq L \leq M$ . MinSpace is the minimum amount of available space left to make reallocation worth while. For example if we assume MinSpace = 1, then we use up the last available unit of memory prior to reporting failure. It would be more realistic to set MinSpace to say 10% of M. If available memory drops below a reasonable level, overflow and re-packing will occur repeatedly. The resulting overhead would be unacceptable in many applications. While every unit of memory can be utilized by this algorithm, such use is normally unreasonable.

**ReA1:** *{Find the amount of available space for reallocation. TotalInc is the total growth since the last time memory overflowed.}*

```
AvailSpace := M - L0 ( or Base[N+1] - Base[0] ); TotalInc := 0; J := N;
While J > 0 Loop
    AvailSpace := AvailSpace - (Top[J] - Base[J]);
    If Top[J] > OldTop[J] then
        Growth[J] := Top[J] - OldTop[J];
        TotalInc := TotalInc + Growth[J];
    Else
        Growth[J] := 0;
    End If;
    J := J - 1;
End Loop;
```

**ReA2:** *If AvailSpace < (MinSpace - 1) then report insufficient memory for re-packing to occur and terminate. {Please note that when memory overflowed, space for the new request has already been reserved at the location pointed to by Top[K]. If AvailSpace = 0, then there is exactly one unit of space available. If AvailSpace < 0, we are truly out of memory.}*

**ReA3:** *GrowthAllocate := 1 - EqualAllocate; {EqualAllocate must be represented as a decimal fraction, e.g., 0.15 would imply 15% of available memory to be allocated equally between all N stacks}*

*Alpha := EqualAllocate \* AvailSpace / N; {EqualAllocate \* AvailSpace is the amount of memory to be divided equally between the stacks. Each stack gets 1/N of this space. Alpha is a real number and must be computed to a reasonable number of digits to the right of the decimal point.}*

**ReA4:** *Beta := GrowthAllocate \* AvailSpace / TotalInc; {TotalGrowthSpace := GrowthAllocate \* AvailSpace is the amount of memory to be allocated based on growth. Beta := TotalGrowthSpace / TotalInc is the amount of space to allocate a stack for each unit it has increased in size since the last time memory overflowed. This algorithm does not penalize for a stack shrinking in size. There is simply no growth allocation, only the equal allocation represented by Alpha. Beta is a real number and must be computed to reasonable accuracy}*

**ReA5:** *NewBase[1] := Base[1]; Sigma := 0;*  
*For J := 2..N Loop {Increase J from 2 through N in increments of 1.}*  
*Tau := Sigma + Alpha + Growth[J-1]\*Beta;*  
*NewBase[J] := NewBase[J-1] + (Top[J-1] - Base[J-1]) + floor(Tau) - floor(Sigma);*  
*Sigma := Tau;*  
*End Loop;*  
*{NewBase[J] is the sum of the preceding base location (Base[j-1]) plus the size of the preceding stack (Top[J-1] - Base[J-1]) plus the growth allocation for the preceding stack (floor(Tau)) minus floor(Sigma)). The "floor" operator means to take the integer part of the number without rounding. For example: floor(0.345) = 0; floor(8.43) = 8;*

*and  $\text{floor}(8.999999) = 8$ . When this step is complete,  $\text{NewBase}[J]$  represents the location of the new base for each stack  $J$  after re-packing.}*

**ReA6:** {It is time to re-pack the stacks. Recall that space for a new element was allocated in a stack pointed to by  $\text{Top}[K]$  when overflow occurred. We must adjust for this prior to re-packing by subtracting one from  $K$ . After re-packing is complete, we must add one back to  $K$  prior to returning to the program requesting the reallocation. The program that requested reallocation of memory may then store the desired datum in the stack location pointed to by  $\text{Top}[K]$ .}

```
Top[K] := Top[K] - 1;  
Perform Algorithm MoveStack;  
Top[K] := Top[K] + 1;  
Insert item causing the over flow at location Top[K];  
For J in 1..N Loop //get ready for next potential overflow.  
    OldTop[J] := Top[J];  
End Loop;
```

### Algorithm MoveStack:

This algorithm moves the contents of each stack. Stacks are moved downward without overlapping stacks to be moved upward or stacks that are to remain in their current location. Note  $\text{Base}[1]$  for stack one should never move. It cannot move below location 1. If moved to location  $C$ , where  $C > 1$ , the locations 1 to  $(C-1)$  would be inaccessible (wasted).

**MoA1:** *{Moves all stacks down first. While not obvious, some thought should convenience you this can be done without overlapping any stacks.}*

```
For J in 2..N Loop  
    If NewBase[J] < Base[J] then  
        Delta := Base[J] - NewBase[J];  
        For L in (Base[J] + 1), (Base[J] + 2) .. Top[J] Loop  
            StackSpace[L - Delta] := StackSpace[L];  
        End Loop;  
        Base[J] := NewBase[J];  
        Top[J] := Top[J] - Delta;  
    End If;  
End Loop;
```

**MoA2:** {Move all stacks up without overlapping any stacks}

```

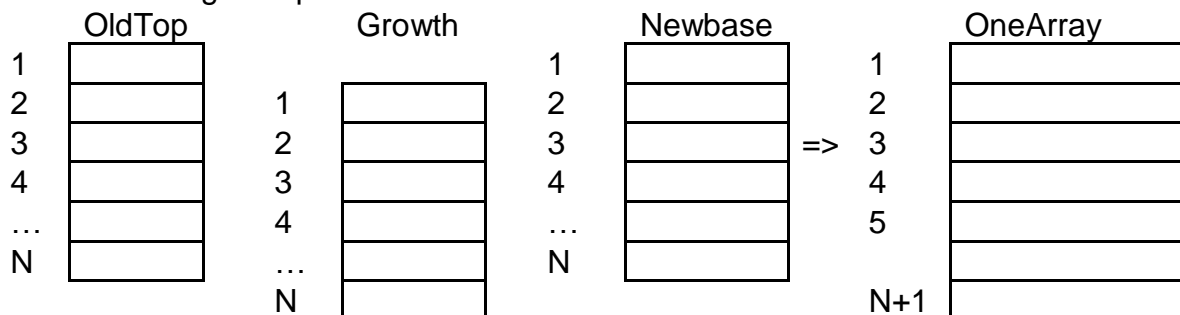
For J in N..2 Loop {J must decrease from N through 2 in steps of 1.}
  If NewBase[J] > Base[J] then
    Delta := NewBase[J] - Base[J];
    For L in Top[J], (Top[J] - 1), ... (Base[J] + 1) Loop
      StackSpace[L + Delta] := StackSpace[L];
    End Loop;
    Base[J] := NewBase[J];
    Top[J] := Top[J] + Delta;
  End If;
End Loop;

```

It is not known who first developed this algorithm. J. Dunlap apparently developed and used these ideas in a series of compilers in 1963. These techniques were also used by IBM during the same time period in a compiler. The techniques were developed separately and first published by Jan Garwick of Norway (BIT 4 (1964), 137-140). Donald Knuth popularized the algorithm in "Fundamental Algorithms," Addison-Wesley, pp. 243-248. These algorithms may be readily extended to any sequentially allocated list. No accurate analysis of run time is known to the author for algorithm Reallocate. Several attempts have been made by various authors to evaluate the run time of algorithm Repack. With simplifying assumptions, all authors approximate the run time proportional to the square of the number of items currently in the stacks. Hence a great deal of time must be spent in moving items if we wish to utilize all units of memory. **This is a classic space-time tradeoff.** The algorithm is well behaved when memory is only 50% full. That is, very little time is spent in re-packing. When possible, memory utilization should probably not exceed 75%. Note the largest stack if known should be the first stack as it is never moved. This will reduce overhead.

**Space Optimization:** The arrays OldTop, Growth, and Newbase can occupy the same physical space (OneArray): **OldTop[J] = Growth[J-1] = Newbase[J]** for  $1 \leq J \leq N + 1$ .

Once values from OldTop and Growth are used to calculate a newbase the oldtop value is no longer required.





Assume the following StackSpace using locations 1 through 100, M= 100.

|   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |              |              |     |     |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|--------------|--------------|-----|-----|
|   | A | B | C | D | E |   |   |   |   |    | A  | B  | C  | D  | E  | F  | <del>A</del> | <del>B</del> | ... |     |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17           | 18           | 19  | 100 |

In the beginning:

|          |                |               |             |
|----------|----------------|---------------|-------------|
| Original | OldTop(1) = 0  | Base(1) = 0   | Top(1) = 0  |
| Values   | OldTop(2) = 10 | Base(2) = 10  | Top(2) = 10 |
|          | OldTop(3) = 16 | Base(3) = 16  | Top(3) = 16 |
|          |                | Base(4) = 100 |             |

The stack space appears as above after processing the following operations. Note the operation **I2G** causes stack 2 to overflow into stack 3 which is currently empty. "I1A" should be interpreted as insert an "A" in stack 1. "D3" should be interpreted as delete the top item in stack 3.

|     |     |     |     |     |             |     |     |     |     |
|-----|-----|-----|-----|-----|-------------|-----|-----|-----|-----|
| I1A | I1B | I1C | I1D | I1E | I3A         | I3B | I2A | I2B | I2C |
| I2D | I2E | I2F | D3  | D3  | <b>I2G*</b> |     |     |     |     |

At overflow:

|               |                                                     |
|---------------|-----------------------------------------------------|
| Base(1) = 0   | Top(1) = <del>0, 1, 2, 3, 4, 5</del>                |
| Base(2) = 10  | Top(2) = <del>10, 11, 12, 13, 14, 15, 16</del> , 17 |
| Base(3) = 16  | Top(3) = <del>16, 17, 18, 17, 16</del>              |
| Base(4) = 100 |                                                     |

**ReA1:**

$L_0 < L \leq M \Rightarrow 0 < L \leq 100$

AvailSpace := M - L<sub>0</sub> := M - 0 := 100.

TotalInc := 0

The number of stacks is N := 3.

Hence:      AvailSpace := AvailSpace - (5-0) - (17 - 10) - (16 - 16) = 88.

Growth[1] := 5 - 0 := 5.

Growth[2] := 17 - 10 := 7.

Growth[3] := 16 - 16 := 0.

TotalInc := 5 + 7 + 0 := 12.

**ReA2:**

Note AvailSpace = 88 implies there are 88 spaces to allocate not including the space required to store "G" into stack 2. That space was reserved at the time overflow occurred.

**ReA3:**

Assume EqualAllocate := 0.1 or 10% equal space allocation. Then GrowthAllocate :=  
 $1.0 - 0.1 := 0.9$  or 90% of space allocated based on dynamic growth.

**Alpha** :=  $0.1 * (\text{AvailSpace} / N) := 0.1 * (88.0 / 3.0) := \mathbf{2.9333}$ .

**ReA4:**

**Beta** :=  $0.9 * (\text{AvailSpace} / \text{TotalInc}) := 0.9 * (88.0 / 12.0) := \mathbf{6.5999}$ .

**ReA5:**

**{J := 1}**

Do nothing. The base of stack 1 never moves.

**NewBase[1] := 0;**

Sigma := 0;

+++++

**{J := 2}**

Tau := Sigma + Alpha + Growth[1]\*Beta :=  $0 + 2.9333 + 5 * 6.5999 := 35.933$ ;

**NewBase[2] := NewBase[1] + Top[1] - Base[1] + floor(Tau) - floor(Sigma)**

**:=  $0 + 5 - 0 + 35 - 0 := \mathbf{40}$ ;**

**Hence NewBase[2] := 40;**

Sigma := Tau := 35.9333

+++++

**{J := 3}**

Tau := Sigma + Alpha + Growth[2]\*Beta :=  $35.9333 + 2.9333 + 7 * 6.5999 := 85.0666$ ;

**NewBase[3] := NewBase[2] + Top[2] - Base[2] + floor(Tau) - floor(Sigma)**

**:=  $40 + 17 - 10 + 85 - 35 := \mathbf{97}$ ;**

**Hence NewBase[3] := 97;**

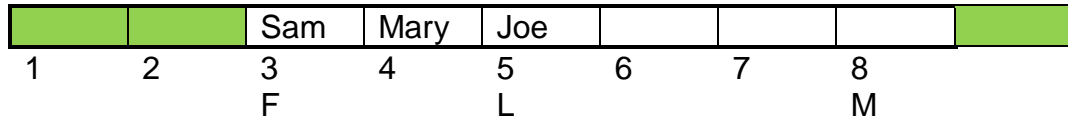
Sigma := Tau := 85.9333;

+++++

We are finished with algorithm ReAllocate and ready to repack memory!

## Basic Sequentially Allocated List Operations

### Bounded Linearly Allocated List



Assume a sequentially allocated list occupy memory locations F(first item) through M (maximum location). Further assume the last item in the list currently occupies location L (current last item). The following are basic list operations:

#### Insert a new item as the last entry in the list:

- 1) Prompt user for "y" to insert in the list.
- 2) If  $L < M$   
     $L \leftarrow L + 1$   
    List[L]  $\leftarrow$  y  
Else  
    Report the list is full, overflow.  
End if

#### Randomly Insert a new item in the list at location J, $J \geq F$ and $J \leq L$ :

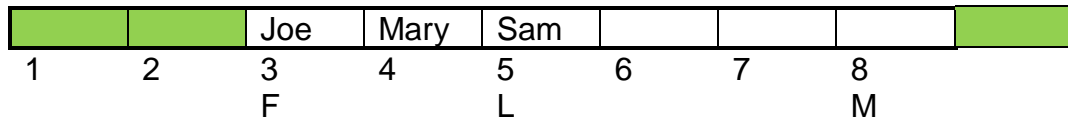
- 1) Prompt the user for a valid value of J.
- 2) If  $(L < M)$  // there is space left in the list  
    For K := L down to J loop  
        List[ K+1 ]  $\leftarrow$  List[ K ];  
    End Loop;  
    List[ J ]  $\leftarrow$  y;  
    L  $\leftarrow$  L + 1;  
Else  
    Overflow;  
End if;

#### Randomly Delete the Jth item from the list, $J \geq F$ and $J \leq L$ :

- 1) Prompt the user for a valid value of J.
- 2) For K := J to (L-1) loop  
    List[ K ]  $\leftarrow$  List[ K+1 ];  
End loop;  
L  $\leftarrow$  L - 1;

\*\* In general random insertion and deletion require moving approximately half the list. Ther worst case for a list with N items is moving N-1 items!

### Bounded Linearly Allocated List



Assume a sequentially allocated list occupies memory locations F(first item) through M (maximum location). Further assume the last item in the list currently occupies location L (current last item) and there is at least one free place in the list ( $L < M$ ). The following are basic list operations:

#### Insert a new item in lexicographic (ascending sorted order, no check for overflow; written for clarity):

```

1) Prompt the user for the value to insert y.
2) K := L;
   While ( K >= F and List[ K ] > y ) loop
       List[ K+1 ] <-- List[K];
       K <-- K + 1;
   End loop;

   If ( K = L )           // New last item in list
       List[ L+1 ] <-- y
   Else If ( K < F )      // New first item in list
       List[ F ] <-- y;
   Else                  // New interior item in list
       List[ K + 1 ] <-- y;
   End if;

   L <-- L + 1;

```

New values of “y” are inserted behind all existing values of “y.” We assume the Boolean short circuit for the test “While ( K >= F and List[ K ] > y ) loop”, i.e., if “K >= F” is false, the second test “List[ K ] > y” is not performed.

\*\* In general random insertion and deletion in a sorted list requires moving approximately half the list. The worst case for a list with N items is moving the entire existing N items!

Strategy for languages like Ruby and Groovy (type declaration not required):

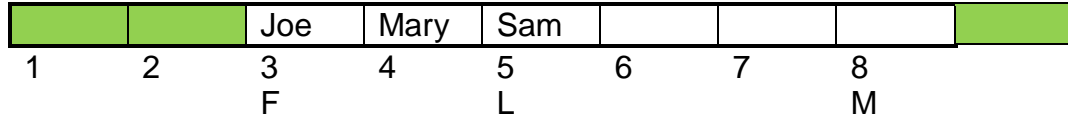
**list1 = [ 'Abc', 15.6, 'Tom', 'Bob'];**

**var1 = 345;**

**list2 = [ 'Mary', \$(Var1), 'Sam', \*list1, 'Betty'];** where “\*” is the expansion operator.

**list2[ 2..4 ] = NULL;** where “..” is range to delete from list.

## Bounded Linearly Allocated List



Assume a sequentially allocated list occupies memory locations F(first item) through M (maximum location). Further assume the last item in the list currently occupies location L (current last item) and there is at least one free place in the list ( $L < M$ ). **At the expense of clarity, does the following insert in lexicographic order? Is it faster? How would you handle duplicate values, e.g., and a second person named jow to the list?**

**Insert a new item in lexicographic (ascending sorted order, no check for overflow; minimize code at expense of clarity):**

- 1) Prompt the user for the value to insert y.
- 2) If  $L = M$   
     Report overflow  
   Else  
      $Temp \leftarrow L$
- 3)  $K := L$ ;  
   While ( Not(  $y > List[Temp]$  ) and Not(  $Temp < F$  ) ) loop  
      $List[ tmp + 1 ] \leftarrow List[ tmp ]$ ;  
      $Temp \leftarrow Temp - 1$ ;  
   End loop;  
  
    $List[ Temp + 1 ] \leftarrow y$ ;  
    $L \leftarrow L + 1$ ;

Most people have trouble with negative logic (use of not). If possible always modify negative logic to positive logic. Use of positive logic make it eaiser for average developers to follow logic.

**Insert a new item in lexicographic (ascending sorted order, no check for overflow; minimize code at expense of clarity):**

- 1) Prompt the user for the value to insert y.
- 2) If  $L = M$   
     Report overflow  
   Else  
      $Temp \leftarrow L$
- 3)  $K := L$ ;  
   While (  $Temp \geq F$  and  $y < List[Temp]$  ) loop  
      $List[ Temp + 1 ] \leftarrow List[ Temp ]$ ;  
      $Temp \leftarrow Temp - 1$ ;  
   End loop;  
  
    $List[ Temp + 1 ] \leftarrow y$ ;  
    $L \leftarrow L + 1$ ;

## Stack implemented as a link allocated linear list.

Assume Top =  $\Omega$  indicates the empty stack / list.

Node Format:

|             |      |
|-------------|------|
| Information | Link |
|-------------|------|

Typical Stack:



### Push(Y: in Data, Inserted: out Boolean)

- 1) Pt  $\leftarrow$  Avail -- Get a node of available storage pointed to by Pt (new, malloc, etc.)
- 2) If (successful in getting a node) then // for many languages: pt  $\neq$  NULL  
    Inserted  $\leftarrow$  true;  
    Pt.Info  $\leftarrow$  Y; -- or Info(Pt)  $\leftarrow$  Y;  
    Pt.Link  $\leftarrow$  Top; -- or Link(Pt)  $\leftarrow$  Top;  
    Top  $\leftarrow$  Pt;  
Else  
    Inserted := false -- Overflow  
End If;

### Pop(Y: out Data, Poped: out Boolean)

- If Top =  $\Omega$  then  
    Poped  $\leftarrow$  false; -- Report underflow.  
Else  
    Pt  $\leftarrow$  Top;  
    Y  $\leftarrow$  Top.Info; -- or Y  $\leftarrow$  Info(Top);  
    Top  $\leftarrow$  Top.Link; - or Top  $\leftarrow$  Link(Top);  
    Pt  $\Rightarrow$  Avail; --Avoid memory hemorrhaging.  
End If;

\*\*\*\*\*



Assume the available storage list itself is maintained as a linked list:

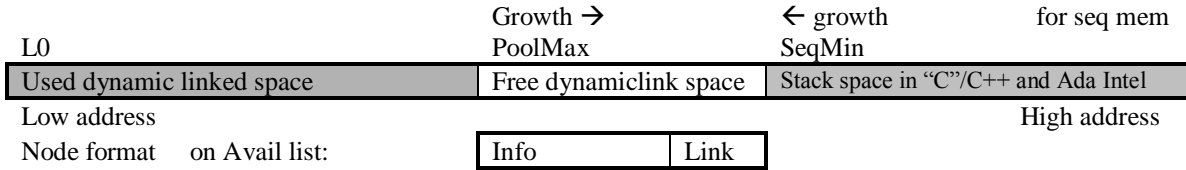
### Pt $\leftarrow$ Avail

- If Avail  $\neq$   $\Omega$  then      -- There is available storage.  
    Pt  $\leftarrow$  Avail;      Avail  $\leftarrow$  Avail.Link // or Avail  $\leftarrow$  Link(Avail);  
Else  
    Report overflow has occurred, e.g., Pt :=  $\Omega$   
End If;

### Pt $\Rightarrow$ Avail

- 1) Pt.Link  $\leftarrow$  Avail      // or Link(Pt)  $\leftarrow$  Avail;
- 2) Avail  $\leftarrow$  Pt;      // or Avail  $\leftarrow$  Pt

# Dynamic Storage Allocation Algorithm



Avail :=  $\Omega$ , if there are no currently available nodes, otherwise Avail points to the first available node on the list.

Assume an area of contiguous memory locations that we would like to use for dynamic storage allocation. We will assume for convenience that all nodes to be allocated are of size "c." We further assume that it is desirable to reuse space previously allocated for dynamic storage allocation that has been returned to the storage pool prior to allocating new storage. We will also assume the existence of a sequentially allocated portion of memory above/below the dynamically allocated portion in which multiple sequentially allocated data structures are maintained. The sequentially allocated portion may be managed by algorithms allowing for re-packing. Several commercial operating systems have used this algorithm as the basis for allocating main memory to processes.

Based on Knuth, Vol. 1, *Fundamental Algorithms*, page 254.

Initially set Avail to  $\Omega$  and PoolMax to  $L_0$ , the base address of the storage pool. SeqMin marks the start of storage not available for dynamic allocation. For many operating systems it marks the top of the stack growing towards Poolmax (at a lower memory address on the Intel architecture).

On each request for memory, the operation **Pt <= Avail** becomes:

```

If Avail /=  $\Omega$ , then
    Pt ← Avail,  Avail ← Avail.Link // or Avail := Link(Avail)
else
    PoolMax ← PoolMax + c where "c" is the desired node size,
    if PoolMax > SeqMin then
        report overflow, e.g., Pt ←  $\Omega$ 
    else
        Pt ← PoolMax - c
    end if
end if

```

When other parts of the program attempt to decrease the value of SeqMin, they should signal overflow if SeqMin < PoolMax.

The operation **Pt => Avail** (or **Avail <= Pt**) is:

- 1) Pt.Link <-- Avail // or Link(Pt) ← Avail;
- 2) Avail ← Pt;

## Implementation Examples

| Problem                                                                                                                                     | Document                 | Page    |  |
|---------------------------------------------------------------------------------------------------------------------------------------------|--------------------------|---------|--|
| Abstract Stack Examples – CompStk1.ads                                                                                                      | Data Structures Programs | 59 - 62 |  |
| Passing generic I/O routines                                                                                                                | Data Structures          | 166-168 |  |
| Passing generic I/O routines GIOEX                                                                                                          | Data Structures Programs | 76 - 79 |  |
| Basic Inheritance – tagged1                                                                                                                 | Data Structures Programs | 63      |  |
| Creating Hetrogeneous data structures using inheritance – AbstStck – also know as composition (bottom-up) versus classification ((top-down) | Data Structures Programs | 64-68   |  |

```
type person;  
type personPt is access person;  
type person is record          -- Assume 12 characters of storage.  
    name: string(1..4);  
    age: integer;  
    next: personPt;  
end record;
```

```
pt1, pt2: personPt;  
pt1 := new person'("Joe", 18, null);  
pt1 := new person'("Sam", 25, null); -- hemorage Joe; pt1 := new person'("Sam", 25, pt1);
```

```
pt1 := new person; -- hemorage Sam
pt1.name := "Jill";
pt1.age := 19;
```

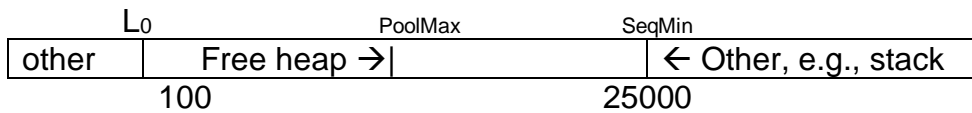
```
&&&&&&&&&&&&&&&&&&&&&&
with Unchecked_Deallocation;
-- pragma Controlled(personPt); --Seldom required, tell compiler we will do garbage collections
-- In code body or specification
procedure Free is new Unchecked_Deallocation( person, personPt);
procedure Free is new Unchecked_Deallocation( car, carPt);
```

```
pt1: personPt := new person'("Tom", 18, null);
```

```
Free(pt1);
```

**Most languages keep a separate Avail list for each fixed data requirement, e.g. person and cars.** This allows fast allocation and reclamation reducing garbage collection overhead. It also means you can run out of storage for one data type and fail even though free memory still exists for the other data type.





```

type person;
type personPt is access person;
type person is record      -- Assume 12 characters of storage, 32 bit architecture.
    name: string(1..4);
    age: integer;
    next: personPt;
end record;

```

L<sub>0</sub> := 100, PoolMax := 100, Avail =  $\Omega$ , SeqMin = 25000

| operation              | PoolMax += C   | allocate  | Avail    |
|------------------------|----------------|-----------|----------|
| New Joe, 18 $\Omega$   | 100 + 12 = 112 | Pt <- 100 | $\Omega$ |
| New Tim 37, $\Omega$   | 112+12=124     | Pt <- 112 | $\Omega$ |
| New Mary, 27, $\Omega$ | 124+12 = 136   | Pt <- 124 | $\Omega$ |
| Delete Tim             | At 112         |           | Avail →  |
| Delete Mary            | At 124         |           | Avail →  |
| New Bob, 42, $\Omega$  | 124            | Pt <- 124 | Avail →  |
|                        |                |           |          |

|      |          |
|------|----------|
| Tim  | $\Omega$ |
| Mary |          |

|     |          |
|-----|----------|
| Tim | $\Omega$ |
| Tim | $\Omega$ |

Initially set Avail to  $\Omega$  and PoolMax to L<sub>0</sub> , the base address of the storage pool. SeqMin marks the start of storage not available for dynamic allocation. For many operating systems it marks the top of the stack growing towards Poolmax (at a lower memory address on the Intel architecture).

On each request for memory, the operation **Pt <= Avail** becomes:

```

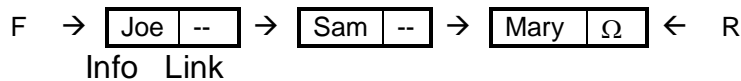
If Avail /=  $\Omega$ , then
    Pt ← Avail,  Avail ← Avail.Link // or Avail := Link(Avail)
else
    PoolMax ← PoolMax + c where "c" is the desired node size,
    if PoolMax > SeqMin then
        report overflow, e.g., Pt ←  $\Omega$ 
    else
        Pt ← PoolMax - c
    end if
end if

```

The operation **Avail <= Pt** or **Pt => Avail** is:

- 1) Pt.Link <-- Avail // or Link(Pt) ← Avail;
- 2) Avail ← Pt;

## Queue implemented as a linked list.



### InsertQueue(Y: in Data, Inserted: out Boolean)

```

1)   Pt <= Avail;
2)   If (successful in getting a node of storage) then
        Inserted := true;
        Pt.Info := Y; -- or Info(Pt) := Y;
        Pt.Link :=  $\Omega$ ; -- or Link(Pt) :=  $\Omega$ ;
        R.Link := Pt; -- or Link(R) := Pt;
        R := Pt;
    Else
        Inserted := false;
    End If;

```

### Boundary conditions:

We desire the above to work for all lists including the empty list to avoid special cases. This can be accomplished by setting  $F := \Omega$  and setting  $R := \text{Location}(F)$  when empty. Note that  $F = \text{Link}(\text{Location}(F))$ . Location {LOC} is interpreted as a physical or logical location for a node.

| Information      | Link          |                           |
|------------------|---------------|---------------------------|
|                  | $F := \Omega$ | Condition for empty list! |
| Location: 123    |               |                           |
| Hence $R := 123$ |               |                           |

This wastes a node of storage. F must be stored as the link field of its location. It allows for run time efficiency however as it eliminates tests for special cases on insertion into the queue (the empty queue). Care must be taken however on deleting the last node. The space in the head node is typically used to store information about the list and not really lost.

The loss of space in the node containing "F" may be avoided by maintaining a count of the number of nodes in the list to check for underflow and & overflow at the expense of run time.

### DeleteQueue(Y: out Data, Removed: out Boolean)

```


If (F =  $\Omega$ ) then
    Removed := false; -- Underflow
Else
    Removed := true;
    Pt := F
    F := Pt.Link; -- or F := Link(Pt);
    Y := Pt.Info; -- or Y := Info(Pt);
    Pt => Avail; -- Prevent hemorrhaging.
    If (F =  $\Omega$ ) then
        R := Loc(F); -- Deleted last node. Set queue to empty: R <-- location of F
    End If;
End If;

```

Example.

Info link R  $\leftarrow 1$

|      |             |   |    |   |   |   |          |   |    |    |    |    |    |
|------|-------------|---|----|---|---|---|----------|---|----|----|----|----|----|
| info | F= $\Omega$ |   | 13 |   |   |   | $\Omega$ |   |    |    |    |    | 7  |
| 1    | 2           | 3 | 4  | 5 | 6 | 7 | 8        | 9 | 10 | 11 | 12 | 13 | 14 |


Avail = 3 

- The node format requires 2 units of memory, the “info” field followed by a “link” field.
- Shaded cells are in use elsewhere.
- $F \leftarrow \Omega$  at the start and  $R \leftarrow 1$ , the location of F. F is actually the link field at the location of the first node in the queue.
- Avail  $\leftarrow 3$  initially.

After inserting Joe in the queue, it looks like the following with Avail  $\leftarrow 13$ , R  $\leftarrow 3$ .

Info link R

|      |     |     |          |   |   |   |          |   |    |    |    |    |    |
|------|-----|-----|----------|---|---|---|----------|---|----|----|----|----|----|
| info | F=3 | Joe | $\Omega$ |   |   |   | $\Omega$ |   |    |    |    |    | 7  |
| 1    | 2   | 3   | 4        | 5 | 6 | 7 | 8        | 9 | 10 | 11 | 12 | 13 | 14 |

Avail = 13 

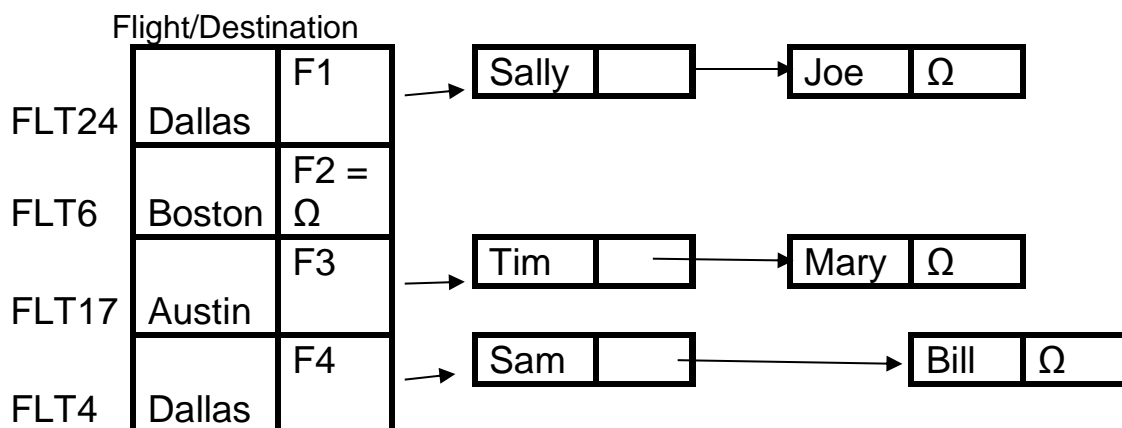
After inserting Sam in the queue, it looks like the following with Avail  $\leftarrow 7$ , R  $\leftarrow 13$

Info link

|      |     |     |    |   |   |   |          |   |    |    |    |     |          |
|------|-----|-----|----|---|---|---|----------|---|----|----|----|-----|----------|
| info | F=3 | Joe | 13 |   |   |   | $\Omega$ |   |    |    |    |     | R        |
| 1    | 2   | 3   | 4  | 5 | 6 | 7 | 8        | 9 | 10 | 11 | 12 | 13  | 14       |
|      |     |     |    |   |   |   |          |   |    |    |    | Sam | $\Omega$ |

Avail = 7 

The “info” field at the location of F is frequently used to store information about the list such as its current length. The following is an airport flight schedule as queues with **list heads**. The flight number is the index into the array of flights.



```

with Ada.Text_IO; use Ada.Text_IO;
with Ada.Unchecked_Deallocation;
procedure LinkQ is
  type JobType is (Programmer, Manager, Accountant, Analyst,
    Sales, Manufacturing, Inventory, SoftwareEngineer);
  package JobTypeIO is new Ada.Text_IO Enumeration_IO(JobType); use JobTypeIO;

  type EmpName is (David, Kevin, Sam, Mary, Bob, Marty, Betty,
    Tom, Teddy, Jerry, Ben, Sara, Donald, Damon, Darlene,
    Dustin);
  package EmpNameIO is new Ada.Text_IO Enumeration_IO(EmpName); use
  EmpNameIO;

  type LegalResponse is (yup, affirmative, Yes, nope, negative, No);
  subtype PositiveResponse is LegalResponse range yup..Yes;
  package LegalIO is new Ada.Text_IO Enumeration_IO(LegalResponse); use
  LegalIO;

  package IntIO is new Ada.Text_IO.Integer_IO(Integer); use IntIO;

  type Emp;
  type EmpPt is access Emp;
  type Emp is record
    Name: EmpName;
    Job: JobType;
    Next: EmpPt;
  end record;

```

**procedure Free is new Ada.Unchecked\_Deallocation(Emp, EmpPt);**

```

procedure InsertQ(NameIn: EmpName; Job: JobType;
  Rear: in out EmpPt; Inserted: out Boolean) is
  Pt: EmpPt;
begin
  Pt := new Emp'(NameIn, Job, null); //Contains potential subtle error.
  if Pt /= null then                // Do you see it?
    Inserted := true;
    Rear.Next := Pt;
    Rear := Pt;
  else
    Inserted := false;
  end if;
end InsertQ;

```

```

procedure DeleteQ(NameOut: out EmpName; JobOut: out JobType;
    Front, Rear: in out EmpPt; Removed: out Boolean) is
    Pt: EmpPt;
begin
    if Front.Next = null then
        Removed := false;
    else
        Removed := true;
        Pt := Front.Next;
        Front.Next := Pt.Next;
        NameOut := Pt.Name; JobOut := Pt.Job;
        Free(Pt); -- Prevent memory hemmoraging.
        if Front.Next = null then
            Rear := Front;
        end if;
    end if;
end DeleteQ;

```

**Front: EmpPt := new Emp;** -- Front.Next is the actual "front" of queue.  
**Rear: EmpPt := Front;** -- Set the queue empty initially, Rear <-- loc of F.

```

OperationCompleted: Boolean;
Again: LegalResponce := affirmative;
TName: EmpName;
TJob: JobType;
begin
    while (Again in PositiveResponce) loop
        put("Enter name: "); get(TName); --Get emp info.
        put("Enter Job type: "); get(TJob);
        -- Insert in appropriate list (by job).
        InsertQ(TName, TJob, Rear, OperationCompleted);

        put("Enter another name (yup or nope): "); get(Again);
    end loop;

    while Front.Next /= null loop
        new_line; put("Employee is = ");
        DeleteQ(TName, TJob, Front, Rear, OperationCompleted);
        if OperationCompleted then -- This test is not really necessary.
            put(TName); put(" "); put(TJob); new_line;
        end if;
    end loop;
end LinkQ;

```

## Mr. Spock:

As you know we have just finished a successful fight with three Klingon Birds of Prey. Normally all Klingons choose to terminate with their ship. Occasionally however, as in this battle, some Klingons chose to escape in life pods. You are hereby authorized to use the transporters to transfer captured Klingons into jail cells dynamically created utilizing the matter-anti-matter pods. Please place the cells in a linked list utilizing the external "hook" on the Enterprise. After capturing the required number of prisoners, you may process them at will with your mind probe once we clear this quadrant. Please avoid vegetating as many minds as possible during the interrogations.

Captain Kirk.

## Mission Impossible

Mr. Phelps,

Your mission, should you decide to accept is to sort agents by mission category. We currently employ 10,000 agents world wide that must be sorted into 100 job categories. There is a catch. To avoid detection on the net, you may not examine any employee record more than once during the sorting process. You may examine each record a second time to produce a listing sorted by job category. After the listing has been produced, the records should still be in sorted order in case further processing is required. This message will self destruct in 10 seconds.

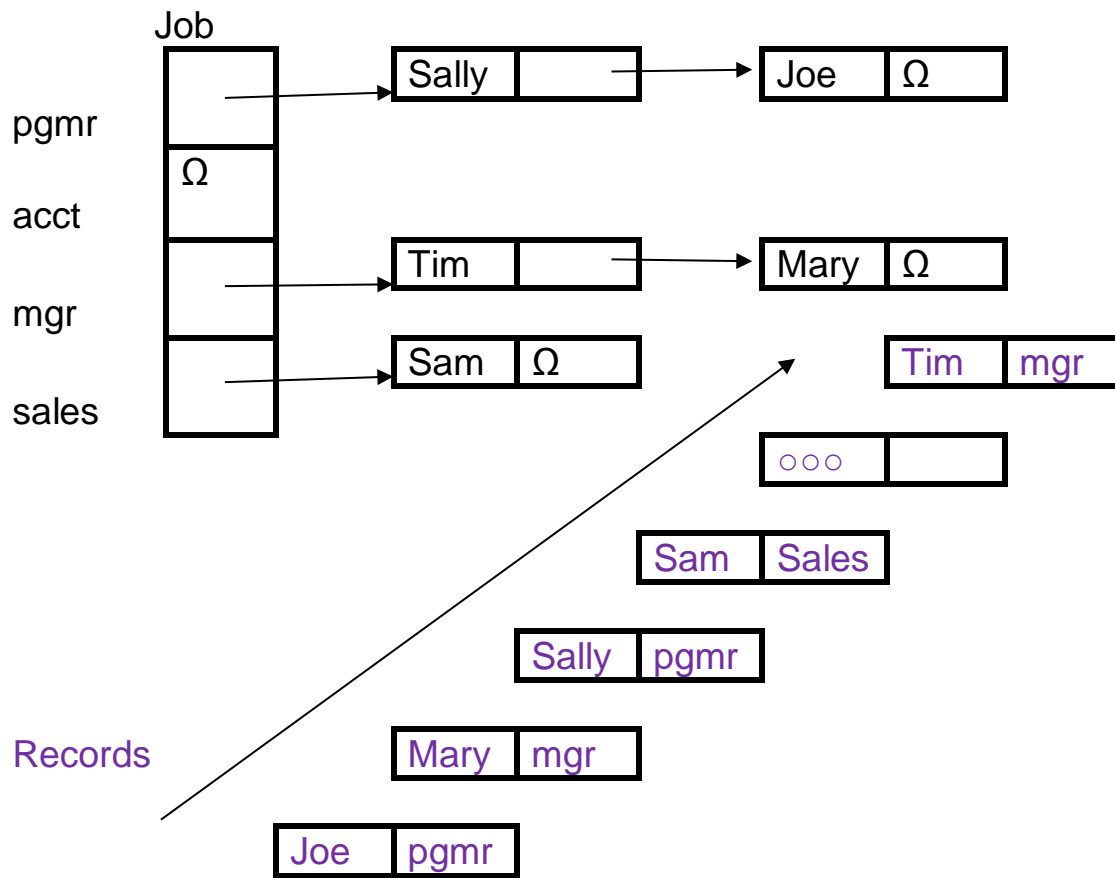
Ex: Joe, pgmr; Mary, mgr; Sally, pgmr; ..., Sam, sales;

## Mission Impossible

Mr. Phelps,

Your mission is to design and code a directory listing all agents world wide currently utilized by the "Impossible Missions Force." Please be sure to minimize the time required to locate agents (by name). Also minimize the effort required to add new agents to the list. Assume 26,000 agents at the start evenly distributed over the alphabet. Please inform management of the minimum number of probes to find an agent, the expected number of probes, and the maximum number of probes under the stated assumptions. What would be the worst case number of probes to find an agent if the distributions assumptions are not adhered to?

## Mission Impossible - Sorting:



Sort time by job category is proportional to the number of records  $N$ ,  $O(N)$ .

The "Job" array is sequentially allocated to take advantage of the ability to locate the list head via direct random access.

The list nodes are dynamically allocated to minimize storage as we do not know ahead of time how many people will be on a given list. A 2-dimensional array would require  $\# \text{ jobs} * \# \text{ people}$  space. The above strategy only requires  $\# \text{ jobs}$  sequential allocation +  $\# \text{ people}$  dynamic nodes.

Consider the business implication of using a stack versus a queue to store the records.

-- Sort a group of 200 employee records by our 8 job categories. You may  
-- not look at any record more than once while placing them in sorted  
-- order. Print the sorted list but do not destroy the sort sequence.

with Ada.Text\_IO; use Ada.Text\_IO;

procedure LinkSort2 is

type JobType is (Pgm, Mgr, Acct, Anal, Sales, Manuf, Inven,  
SoftwareEnginner);

package JobTypeIO is new Ada.Text\_IO.Enumeration\_IO(JobType);  
use JobTypeIO;

type EmpName is (David, Kevin, Sam, Mary, Bob, Marty, Sable, Betty,  
Tom, Teddy, Jerry, Ben, Sara, Donald, Damon, Darlene, Dustin,  
Desire);

package EmpNameIO is new Ada.Text\_IO.Enumeration\_IO(EmpName);  
use EmpNameIO;

type LegalResponse is (yup, affirmative, nope, negative);

subtype PositiveResponse is LegalResponse range yup..affirmative;

package LegalIO is new Ada.Text\_IO.Enumeration\_IO(LegalResponse);  
use LegalIO;

package IntIO is new Ada.Text\_IO.Integer\_IO(Integer); use IntIO;

**type Emp;**

type EmpPt is access Emp;

**type Emp is record** -- or type Emp is tagged record

  Name: EmpName;

  Job: JobType;

  Next: EmpPt;

**end record;**

**SortByJob: Array(JobType) of EmpPt;** -- or Array(Mgr .. Inev) of EmpPt

TempName: EmpName;

TempJob: JobType;

Pt: EmpPt;

Again: LegalResponse := affirmative;



```

begin
  while (Again in PositiveResponse) loop
    put("Enter name (David, Kevin, Sam, Mary, or Bob): ");
    get(TempName);
    put("Enter Job type (Pgmr, Mgr, Acct, Anal, Sales: "); get(TempJob);
    -- Insert in appropriate list (by job).
    SortByJob(TempJob):=
      new Emp'(TempName, TempJob, SortByJob(TempJob));

    put("Enter another name (yup or nope): "); get(Again);
  end loop;

  for I in JobType loop -- Traverse.
    new_line; put("Job Type = "); put (I); new_line;
    Pt := SortByJob(I); -- Point to first node in job list.
    while Pt /= null loop
      put(Pt.Name); put(" "); put(Pt.Job); new_line; Pt := Pt.Next;
      -- Move down list.
    end loop;
  end loop;
end LinkSort2;

```

**Sample Input / Output:**

```

Enter name: David
Enter Job type: Sales
Enter another name (yup or nope): yup
Enter name: Mary
Enter Job type: Programmer
Enter another name (yup or nope): yup
Enter name: Tom
Enter Job type: Manager
Enter another name (yup or nope): yup
Enter name: Ben
Enter Job type: Manufacturing
Enter another name (yup or nope):yup
Enter name: Betty
Enter Job type: Sales
Enter another name (yup or nope): nope

```

```

Job Type = PROGRAMMER
MARY  PROGRAMMER link =      0

```

Job Type = MANAGER  
TOM MANAGER link = 0

Job Type = ACCOUNTANT

Job Type = ANALYSIST

Job Type = SALES  
BETTY SALES link = 1  
DAVID SALES link = 0

Job Type = MANUFACTURING  
BEN MANUFACTURING link = 0

Job Type = INVENTORY

Job Type = SOFTWAREENGINEER

```
-- Sort a group of 200 employee records by our 8 job categories.  You may
-- not look at any record more than once while placing them in sorted
-- order.  Print the sorted list but do not destroy the sort sequence.
```

```
with Ada.Text_IO; use Ada.Text_IO;
```

```
procedure LinkSort is
```

```
  type JobType is (Programmer, Manager, Accountant, Analyst,
                   Sales, Manufacturing, Inventory, SoftwareEngineer);
```

```
  package JobTypeIO is new Ada.Text_IO Enumeration_IO(JobType);
  use JobTypeIO;
```

```
  type EmpName is (David, Kevin, Sam, Mary, Bob, Marty, Sable, Betty,
                   Tom, Teddy, Jerry, Ben, Sara, Donald, Damon, Darlene,
                   Dustin, Desire);
```

```
  package EmpNameIO is new Ada.Text_IO Enumeration_IO(EmpName);
  use EmpNameIO;
```

```
  type LegalResponse is (yup, affirmative, nope, negative);
```

```
  subtype PositiveResponse is LegalResponse range yup..affirmative;
```

```
  package LegalIO is new Ada.Text_IO Enumeration_IO(LegalResponse);
  use LegalIO;
```

```
  package IntIO is new Ada.Text_IO.Integer_IO(Integer); use IntIO;
```

```
  type Emp is record
```

```
    Name: EmpName;  Job: JobType;  Next: integer;
  end record;
```

```
  SortByJob: Array(JobType) of integer := (others => 0);
```

```
  SortSpace: Array(1..200) of Emp;
```

```
  Avail: integer := 1; -- Dynamic storage allocator.
```

```
  Pt: integer;
```

```
  Again: LegalResponse := affirmative;
```

```

begin
  while (Again in PositiveResponse) loop
    put("Enter name: "); get(SortSpace(Avail).Name); --Get emp info.
    put("Enter Job type: "); get(SortSpace(Avail).Job);
    -- Insert in appropriate list (by job).
    SortSpace(Avail).Next := SortByJob(SortSpace(Avail).Job);
    SortByJob(SortSpace(Avail).Job) := Avail;
    -- Prepare for next dynamically allocated node.
    Avail := Avail + 1;

    put("Enter another name (yup or nope): "); get(Again);
  end loop;

  for I in JobType loop -- Traverse.
    new_line; put("Job Type = "); put (I); new_line;
    Pt := SortByJob(I); -- Point to first node in job list.
    while Pt /= 0 loop
      put(SortSpace(Pt).Name); put(" "); put(SortSpace(Pt).Job);
      put(" link = "); put(SortSpace(Pt).Next,4); new_line;
      Pt := SortSpace(Pt).Next; -- Move down list.
    end loop;
  end loop;
end LinkSort;

```

- \* Sort 100 records into their 10 job categories. You may not look
- \* at any record more than one to place it in its final sorted position.
- \* Please list the sorted records. Leave the records in sorted order.
- \*

**Identification Division.**

**Program-ID. Link1.**

**Environment Division.**

**Data Division.**

**Working-Storage Section.**

**01 DynamicStorage.**

10 Node occurs 100 times.

20 NName pic X(20).

20 NJob pic 999.

20 NLink pic 999.

10 Head occurs 10 times pic 999.

10 Avail pic 999.

01 UserResponse pic XXX.

01 NumericResponse pic 9.

01 HitAnyKey pic X.

**01 LinkListVariables.**

10 Pt pic 999 usage is comp.

10 I pic 99 usage is comp.

10 Job pic 99.

**Screen Section.**

**01 MainMenu**

Background-Color is 1

Foreground-Color is 7.

05 value 'Menu' blank screen highlight  
line 02 column 33.

05 value 'A Division of Burris Industries' line 03 column 23.

05 value '1. Insert new data?' line 04 column 05.

05 value '2. Print sorted data?' line 05 column 05.

05 value '3. Exit the program!' line 06 column 05.

05 value 'Please enter your choice>>> ' line 12 column 05.

**Procedure Division.**

**MainSection Section.**

Move zero to DynamicStorage.

Perform varying I from 1 by 1 until I > 10 Move Zero to Head(I).

Move zero to Avail.

Move "No" to UserResponse.

Perform until (UserResponse = "Yes" or UserResponse = "yes")

Display MainMenu

Accept NumericResponse

Evaluate NumericResponse

when 1 Perform InsertRecordSection

when 2 Perform PrintSortedByCategorySection

when 3 Move "Yes" to UserResponse

when other Display "Illegal response, try again!"

Accept HitAnyKey

End-Evaluate

End-Perform.

Stop Run.

**InsertRecordSection Section.**

Display "Insert Record".

Add 1 to Avail.

Display "Enter the employees name: " with no advancing.

Accept NName(Avail).

Display "Enter the job category: " with no advancing.

Accept Job.

Move Job to NJob(Avail).

Move Head(Job) to NLink(Avail).

Move Avail to Head(Job).

**PrintSortedByCategorySection Section.**

Display "Records sorted by category!".

Perform varying I from 1 by 1 until I > 10

Move Head(I) to Pt

Display "List ", I

Perform until Pt = 0

Display " ", NName(Pt)

Move NLink(Pt) to Pt

End-Perform

End-Perform.

Display "Hit any key to continue".

Accept HitAnyKey.

**End Program Link1.**

```

/* in file lsort.cpp, a good method to sort records into related groups */
#include<stdio.h>
#include<stdlib.h> //contains malloc() prototype, the "new" allocator is preferred in C++.
#define NULL 0
struct per_rec {
    char name[10];
    int job;
    struct per_rec *next;
};

void main( ) {
    struct per_rec *head[5] = {NULL,NULL,NULL,NULL,NULL},
    *pt;
    int i;
    char again[10];

    do {
        pt = (per_rec*) malloc(sizeof(per_rec)); //allocate a new node
        // pt = new per_rec; // is preferred in C++
        if (pt == NULL) { //terminate, out of memory
            printf("Out of memory, termination is forced!");
            *again = 'n';
        }
        else { //get the data
            printf("Enter the name(10 chars max): ");
            // cout << "Enter the name(10 chars max)";
            scanf("%s", pt -> name); // cin >> pt -> name;
            printf("Enter the job category (0 thru 4): ");
            scanf("%d", &i);
            pt -> job = i;

            //insert the node in the appropriate job list
            pt -> next = *(head + i); *(head + i) = pt;
        } //end if

        printf("More data? (Y/N): "); scanf("%s", again);
    }
    while( *again == 'y' || *again == 'Y'); //end do-while
}

```

```

// Traverse all the lists printing their contents.
for (i = 0; i < 5; i++)
{
    printf("\n\ngroup %d\n",i);
    //get list head
    pt = head[i];
    //traverse current list.
    while (pt != NULL)
    {
        printf("%s\n",pt -> name);
        pt = pt -> next;
    }
} //end for, get next list head
printf("Thats all folks!");
}

```



## Insert in singly linked list in lexicographic order:

Assume a singly linked list pointed to by Pt. We wish to insert a new node pointed to by lp.

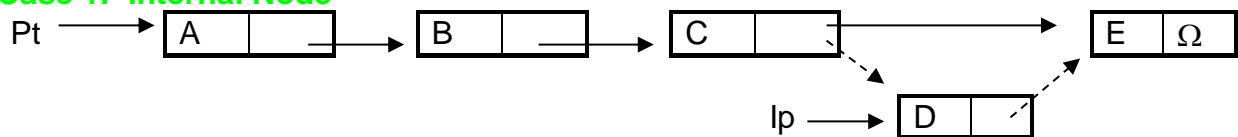
### {find insertion point}

```
P1 <- P2 <- Pt
while (P1 <> null) and (lp.info > P1.Info) loop { left to right}
  P2 <- P1; P1 <- P1.Link
end loop
```

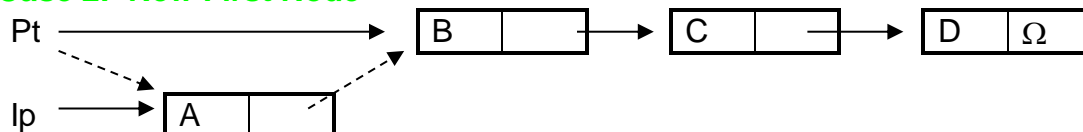
### {insert in list}

```
if Pt = P1 then {new first node or only node}
  lp.Link <- Pt
  Pt <- lp
else {interior node or new last node}
  P2.Link <- lp
  lp.Link <- P1
end if
```

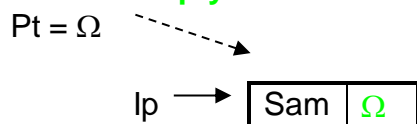
#### Case 1: Internal Node



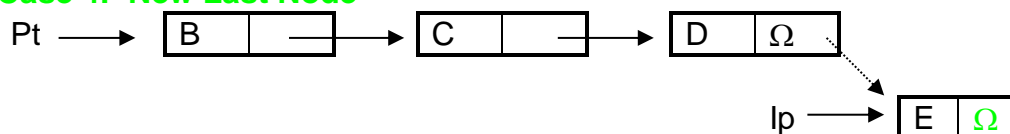
#### Case 2: New First Node



#### Case 3: Empty List



#### Case 4: New Last Node



{Note: If elements are inserted into a list one at a time, this amounts to a sort algorithm}

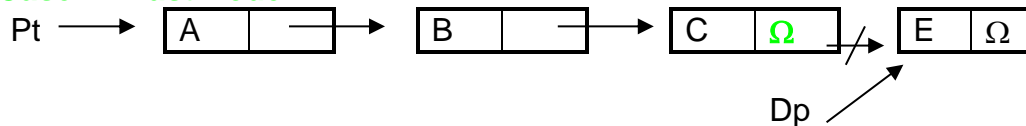
## Delete a random node pointed to by DP from a singly linked list:

```

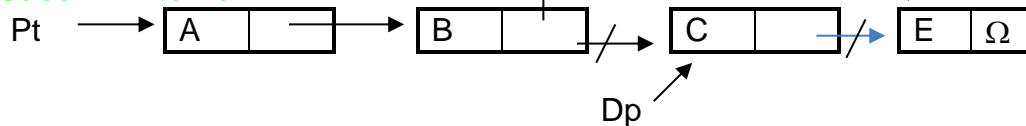
{first node?}
if Pt = Dp then
    Pt <- Pt.Link
    Y <- Dp.Info
    Dp => Avail
else {find node prior to Dp}
    P1 <- Pt
    while P1.Link <> Dp loop
        P1 <- P1.Link
    end loop
    {delete the node}
    P1.Link <- Dp.Link
    y <- Dp.Info
    Dp => Avail
end if

```

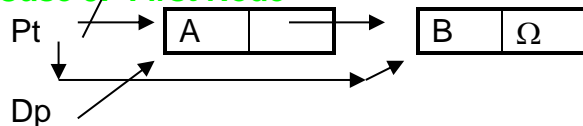
### Case 1: Last Node



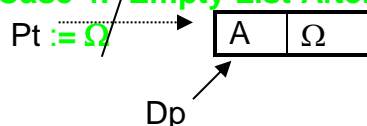
### Case 2: Internal



### Case 3: First Node



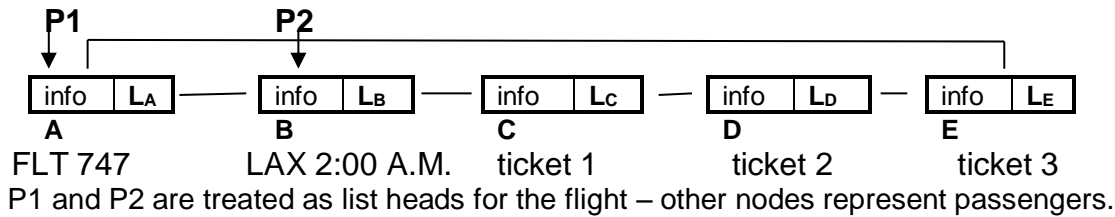
### Case 4: Empty List After Deletion



Inklist1.doc

## Efficient Bi-directional Traversal of a Singly Linked List

Assume a circular list with two consecutive list heads pointed to by pointers P1 and P2 as shown below. Devise a means for completing the link fields such that the list can be traversed efficiently in both directions. That is, if you have pointers to nodes  $X_{i-1}$  and  $X_i$  at addresses A and B, you should be able to directly access the nodes at  $X_{i-2}$  and  $X_{i+1}$  at addresses E and C respectively.



Solution: Let the link fields of node  $X_i$  contain the  $\text{Loc}(X_{i+1}) \oplus \text{Loc}(X_{i-1})$ .

Assume locations “A” and “B” are the locations of the list heads. The pointer fields in the nodes are calculated as follows:

|                     |  |                     |  |                    |
|---------------------|--|---------------------|--|--------------------|
| $L_A = B \oplus E$  |  | $L_B := C \oplus A$ |  | $L_C = D \oplus B$ |
| $L_D := E \oplus C$ |  | $L_E = A \oplus D$  |  |                    |

Where  $\oplus$  is the “exclusive or” operation on the binary representation of the addresses. Recall the truth table for  $\oplus$ :

| $\oplus$ |  | 0 | 1 |
|----------|--|---|---|
|          |  |   |   |
| 0        |  | 0 | 1 |
| 1        |  | 1 | 0 |

**Assume we are at the location of A and B and wish to move forward (to the right):**

Forward :=  $L_B \oplus A \Rightarrow C \oplus A \oplus A \equiv C$ .

**Assume we are at the location of A and B and wish to move backward (to the left):**

Backward :=  $L_A \oplus B \Rightarrow B \oplus E \oplus B \equiv E$ .

**Example:** Assume  $A = 10110$ ,  $B = 01101$ ,  $C = 10010$ ,  $D = 00110$ , and  $E = 11100$ . Then  $L_A = 01101 \oplus 11100 = 10001$ ,  $L_B := C \oplus A = 10110 \oplus 10010 = 00100$ ,  $L_C = 00110 \oplus 01101 = 01011$ ,  $L_D = 10010 \oplus 11100 = 01110$ , and  $L_E = 10110 \oplus 00110 = 10000$ . To move Forward from P2 :=  $L_B \oplus A = 00100 \oplus 10110 = 10010$  which is the desired address “C.”

```
with unchecked_conversion; with system; use system;
with Ada.Text_IO; use Ada.Text_IO;
```

```
procedure uncheck2Address is
```

```
  type ptr is access long_integer;
```

```
  pt: ptr := new long_integer'(46);
```

```
  b: array(1..4) of long_integer := (1,2,3,4);
```

```
  c: address;
```

```
  a: long_integer := 1;
```

```
  package long_int_io is
```

```
    newAda.Text_IO.Integer_IO(long_integer);
```

```
  use long_int_io;
```

```
  function integer_to_address is new
```

```
    unchecked_conversion(long_integer, address);
```

```
  function address_to_integer is new
```

```
    unchecked_conversion(address, long_integer);
```

```
  function integer_to_ptr is new
```

```
    unchecked_Conversion(long_integer, ptr);
```

```
begin
```

```
  c := b'address;
```

```
  for i in 1.. 3 loop
```

```
    a := address_to_integer(c);
```

```
    put(a); put(" ");
```

```
    pt := integer_to_ptr(a);
```

```
    put(pt.all); new_line;
```

```
    a := a + 4; -- for 32 bit addresses
```

```
    c := integer_to_address(a);
```

```
  end loop;
```

```
end uncheck2Address;
```

```
415977414  1
```

```
415977418  2
```

```
415977422  3
```

```
--This is not the only way to do XOR etc on pointers. Package system
-- comes with the data type "address" which allows you to
-- obtain address at run time.
```

```
-- This solution utilizes "modular types" and the fact that the
-- current PC architecture is 32 bit. Since XOR is commutative,
--  $m1 \text{ xor } m2 \text{ xor } m1 = m2$  or Mary in the example. The output is
-- joe
-- Mary
-- Mary
```

```
with Ada.Text_IO; use Ada.Text_IO;
with unchecked_conversion;+*
procedure XOREx is
```

```
type Cell;
type CellPt is access Cell;
```

```
type Cell is record
    Name: String(1..4);
    next: CellPt;
end record;
```

```
type Modular32 is mod 2**32;
```

```
function CellPtToModular is new Unchecked_Conversion(CellPt, Modular32);
function ModularToCellPt is new Unchecked_Conversion(Modular32, CellPt);
```

```
p1, p2, p3: CellPt;
m1, m2, m3: Modular32;
```

```
begin
    p1 := new Cell("joe ", null);
    put( p1.Name ); new_line;
    p2 := new Cell("Mary", null);
    put( p2.Name ); new_line;

    m1 := CellPtToModular (p1 );
    m2 := CellPtToModular( p2 );
    m3 := m1 xor m2 xor m1;

    p3 := ModularToCellPt( m3 );

    put( p3.Name );
end;
```

# Topological Sorting

Sample topological sort. This data set was made famous by Dr. Donald Knuth in the text “Fundamental Algorithms” published by Addison-Wesley.

J < k

9 < 2

3 < 7

7 < 5

5 < 8

8 < 6

4 < 6

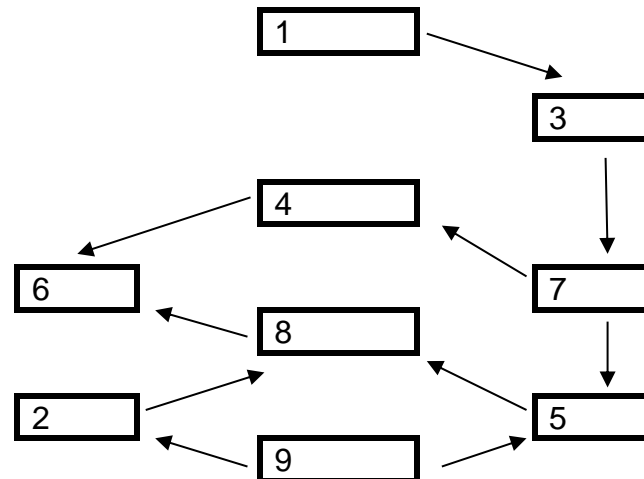
1 < 3

7 < 4

9 < 5

2 < 8

One Solution



Left to right scan:

One possible solution: 1, 3, 7, 4, 9, 2, 5, 8, 6

Using a queue: 1, 9, 3, 2, 7, 4, 5, 8, 6

2<sup>nd</sup> Solution: 9, 2, 1, 3, 7, 5, 8, 4, 6

Topological sorting determines a critical path. Solutions are seldom unique in industry.

Try: 9 < 2, 3 < 7, 7 < 5, 5 < 8, 8 < 6, 4 < 6, 1 < 3, 7 < 4, 9 < 5, 2 < 8

Try a simple left to right scan to select nodes as they become free. Next solve the same problem using a queue.

Try: 1 < 3, 1 < 2, 2 < 6, 2 < 5, 3 < 4, 4 < 5, 5 < 6, 5 < 7, 8 < 7, 9 < 8, and 9 < 6

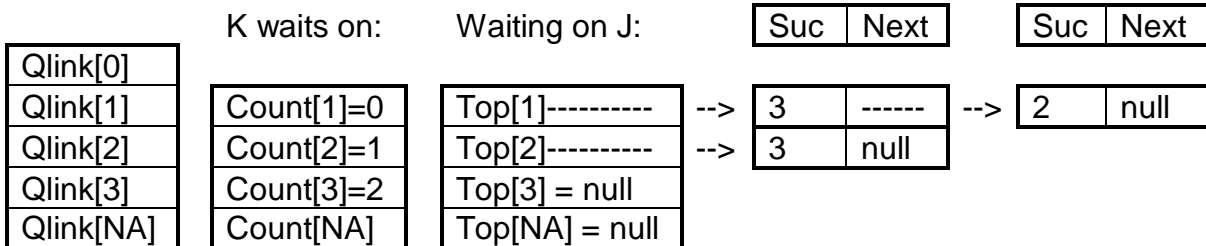
Try: 1 < 2, 2 < 4, 4 < 5, 4 < 6, 5 < 3, and 3 < 2

# Topological Sorting

For the relations “1 < 2,” “1 < 3,” and “2 < 3.”

( “J < K” implies action J precedes action K)

Node format:



## Topological Sort: (Simple – detect existence of loop, not elements)

This algorithm processes relations of the form “J < K” (action J precedes action K) in a partial ordering. It is assumed that the actions are numbered consecutively from 1 to NA. The output of the algorithm is the set of NA actions in topological order (not necessarily unique). The value of Count[K], for 1 ≤ K ≤ NA, represents the number of actions (tasks) that must be accomplished before action K may be performed. The linked list pointed to by Top[J], for 1 ≤ J ≤ NA, represent the actions awaiting the completion of action J prior to their being eligible for output (execution). F and R are used as pointers to the front and rear of a queue of actions eligible for output in a Qlink table. The Qlink and Count may occupy the same space as follows: Qlink[0], Count[1] = Qlink[1], Count[2] = Qlink[2], ..., Count[NA] = Qlink[NA]. A null is indicated by a zero in the link field.

- 1) {initialization}  
Get the number of actions (task to be completed), NA;  
For K in 1.. NA begin Count[K] <- 0; Top[K] <- null; end;  
Set KN <- NA where KN is the number of actions still to be processed.
- 2) {Assume at least one relation. Build the data structure representing which actions “K” must wait on and which actions are waiting on action “J” in the relation “J < K.”}

Notation on right preferred.

|                                                                                                                                                                                                                          |                                                                                                                                                                                                                        |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Get the first relation J < K.<br>Repeat<br>Increase Count[K] by one;<br>Set P <= Avail; Suc[P] <- K;<br>Set Next[P] <- Top[J]; Top[J] <- P;<br>Get the next relation “J < K.”<br>until out of transactions in the input. | Get the first relation J < K.<br>Repeat<br>Increase Count[K] by one;<br>Set P <= Avail; P.Suc <- K;<br>Set P.Next <- Top[J]; Top[J] <- P;<br>Get the next relation “J < K.”<br>until out of transactions in the input. |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

- 3) {Initialize the output queue by linking all Qlink[k] where Count[k] = 0. Count [k] = 0 indicates no task must precede the task K in the output.}  
Set R <- 0 and Qlink[0] <- 0;  
for K in 1.. NA loop  
    If Count[K] = 0 then  
        Qlink[R] <- K; R <- K;  
    end if;  
end loop;  
F <- Qlink[0];
- 4) While F not = 0 loop  
    Perform action F.  
    Set KN <- KN - 1 and P <- Top[F];  
    While P not = null loop  
        Count[Suc(P)] = Count[Suc(P)] - 1;  
        If Count[Suc(P)] = 0 then  
            Qlink[R] <- Suc[P]; {add to output queue}  
            R <- Suc[P]  
        end if  
        P <- Next[P];  
    end loop;  
    F <- Qlink[F];  
end loop;
- 5) If KN = 0, the topological sort has been completed successfully. All actions have been printed in the output stream. If KN is not equal to zero, then the relations have violated the hypothesis for a partial order, i.e., the relations contain a loop.



**J < k**

9 < 2

3 < 7

7 < 5

5 < 8

8 < 6

4 < 6

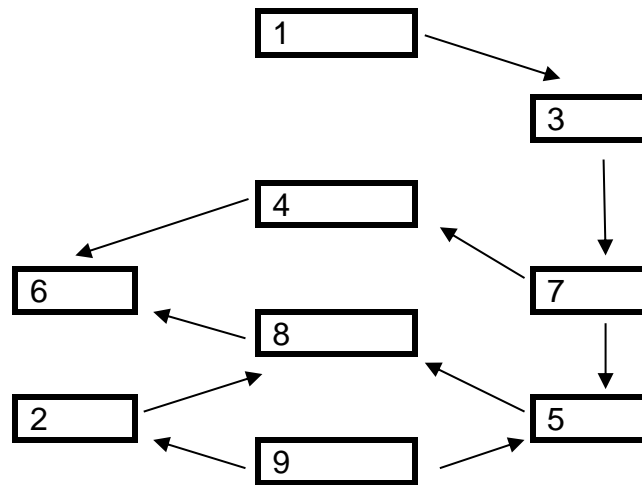
1 < 3

7 < 4

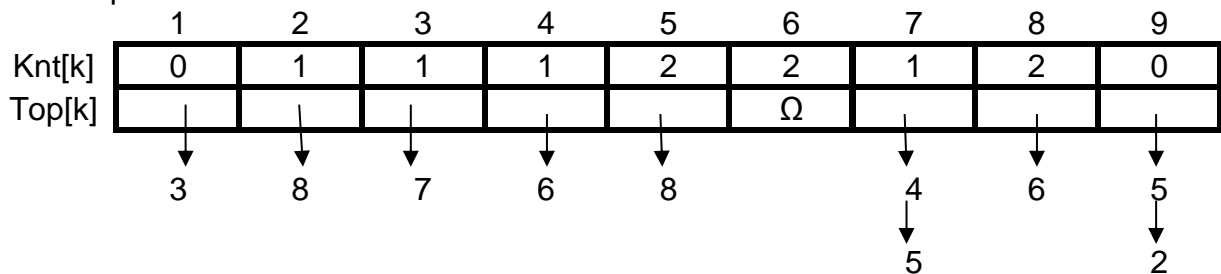
9 < 5

2 < 8

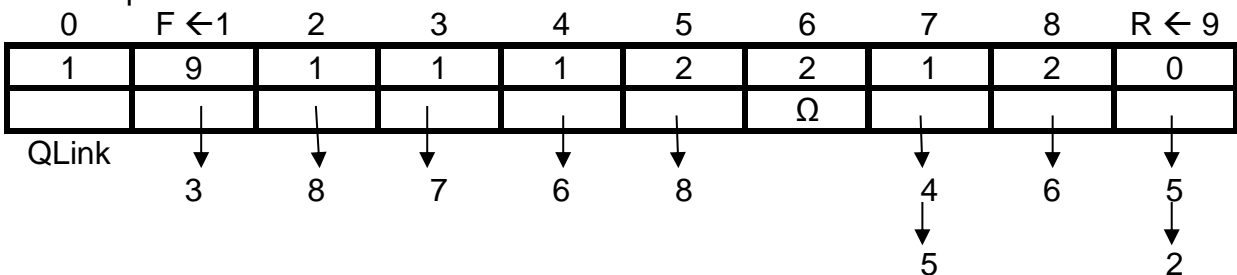
**One Solution**



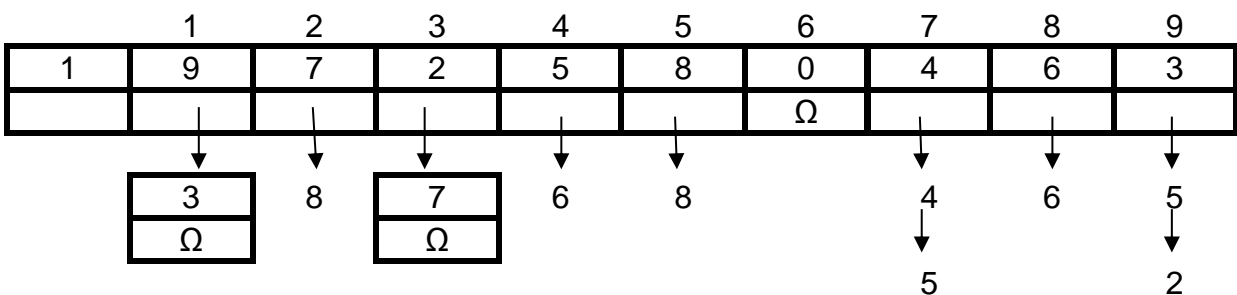
After step 2:



After step 3



After step 4:



F = 1, 9, 3, 2, 7, 4, 5, 8, 6

R = 1, 9, 3, 2, 7, 4, 5, 8, 6, 0

Using a queue: 1, 9, 3, 2, 7, 4, 5, 8, 6

$J < k$

$1 < 3$

$1 < 2$

$2 < 6$

$3 < 4$

$4 < 5$

$5 < 6$

$5 < 7$

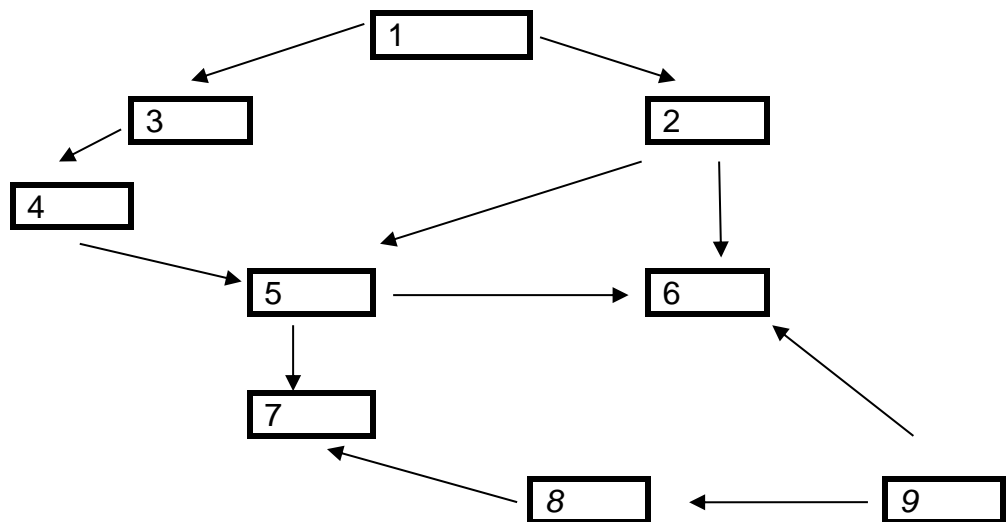
$9 < 8$

$9 < 6$

$2 < 5$

$8 < 7$

One Solution



After step 2:

|        | 1 | 2 | 3 | 4 | 5 | 6        | 7        | 8 | 9 |
|--------|---|---|---|---|---|----------|----------|---|---|
| Knt[k] | 0 | 1 | 1 | 1 | 2 | 3        | 2        | 1 | 0 |
| Top[J] |   |   |   |   |   | $\Omega$ | $\Omega$ |   |   |

Arrows from Top[J] to Knt[k]:  
 1 → 2 → 3, 2 → 5 → 6, 3 → 4, 4 → 5, 5 → 7 → 6, 6 → 7, 7 → 8, 8 → 6

After step 3

| 0 | 1 | 2 | 3 | 4 | 5 | 6        | 7        | 8 | 9 |
|---|---|---|---|---|---|----------|----------|---|---|
| 1 | 9 | 1 | 1 | 1 | 2 | 3        | 2        | 1 | 0 |
|   |   |   |   |   |   | $\Omega$ | $\Omega$ |   |   |

Arrows from Top[J] to Knt[k]:  
 1 → 2 → 3, 2 → 5 → 6, 3 → 4, 4 → 5, 5 → 7 → 6, 6 → 7, 7 → 8, 8 → 6

$F \leftarrow 1$  and  $R \leftarrow 9$

After step 4:

| 0 | 1 | 2 | 3 | 4 | 5 | 6        | 7 | 8 | 9 |
|---|---|---|---|---|---|----------|---|---|---|
| 1 | 9 | 3 | 8 | 5 | 7 | 0        | 6 | 4 | 2 |
|   |   |   |   |   |   | $\Omega$ |   |   |   |

Arrows from Top[J] to Knt[k]:  
 1 → 2 → 3, 2 → 5 → 6, 3 → 4, 4 → 5, 5 → 7 → 6, 6 → 7, 7 → 8, 8 → 6

Final Answer: 1, 9, 2, 3, 8, 4, 5, 7, 6

$J < K$

1 < 2

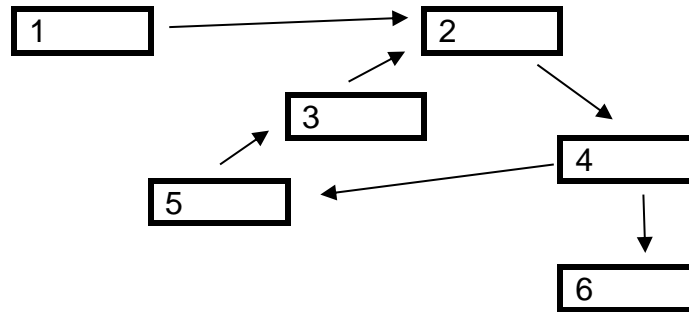
2 < 4

4 < 5

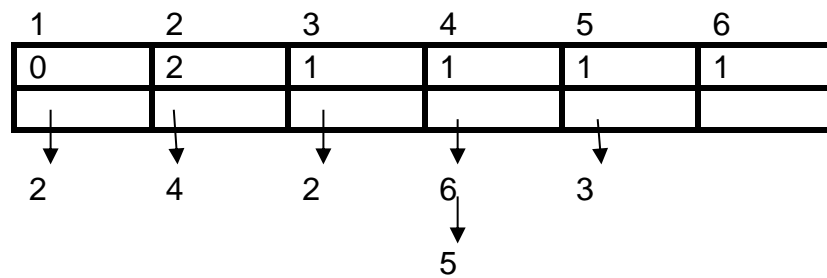
4 < 6

5 < 3

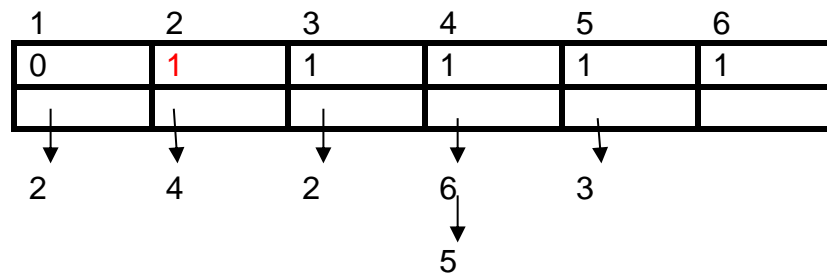
3 < 2



After Step 2:



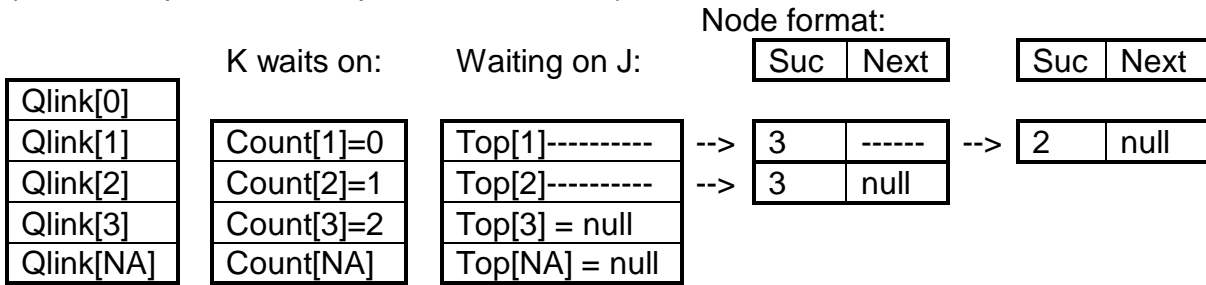
After Step4:



No Solution. Loop shown in data structure: 2 → 4 → 5 → 3 → 2

1) For the relations “1 < 2,” “1 < 3,” and “2 < 3.”

( “J < K” implies action J precedes action K)



**Topological Sort:** This version prints relations that have violated the hypothesis of a partial ordering causing a loop.

This algorithm processes relations of the form “J < K” (action J precedes action K) in a partial ordering. It is assumed that the actions are numbered consecutively from 1 to NA. The output of the algorithm is the set of NA actions in topological order (not necessarily unique). The value of Count[K], for 1 ≤ K ≤ NA, represents the number of actions (tasks) that must be accomplished before action K may be performed. The linked list pointed to by Top[J], for 1 ≤ J ≤ NA, represent the actions awaiting the completion of action J prior to their being eligible for output (execution). F and R are used as pointers to the front and rear of a queue of actions eligible for output in a Qlink table. The Qlink and Count may occupy the same space as follows: Qlink[0], Count[1] = Qlink[1], Count[2] = Qlink[2], ..., Count[NA] = Qlink[NA]. If encountered, the first loop due to an improper set of relations in the partial ordering is printed in reverse order. A null is indicated by a zero in the link field.

- 1) {initialization}  
 Get the number of actions, NA;  
 For K in 1 .. NA begin Count[K] <- 0; Top[K] <- null; end;  
 Set KN <- NA where KN is the number of actions still to be processed.
- 2) {Assume at least one relation. Build the data structure representing which actions “K” must wait on and which actions are waiting on action “J” in the relation “J < K.”}

Get the first relation J < K.

Repeat

Increase Count[K] by one;  
 Set P <- Avail; P.Suc <- K;  
 Set P.Next <- Top[J]; Top[J] <- P;  
 Get the next relation “J < K.”

until out of transactions in the input.

- 3) {Initialize the output queue by linking all Qlink[k] where Count[k] = 0. Count [k] = 0 indicates no task must precede the task K in the output.}  
Set R <- 0 and Qlink[0] <- 0;  
for K in 1 .. NA loop  
    If Count[K] = 0 then  
        Qlink[R] <- K; R <- K;  
    end if;  
end loop;  
F <- Qlink[0];
- 4) While F not = 0 loop  
    Perform action F. {write it in the output}  
    Set KN <- KN - 1; P <- Top[F]; Top[F] <- 0;  
    {Top[I] points to all relations not yet cancelled, 1 <= I <= NA}  
    While P not = 0 loop  
        Count[Suc(P)] = Count[Suc(P)] - 1;  
        If Count[Suc(P)] = 0 then  
            Qlink[R] <- Suc[P]; {add to output queue}  
            R <- Suc[P]  
        end if  
        P <- Next[P];  
    end loop;  
    F <- Qlink[F];  
end loop;
- 5) If KN = 0 then  
    the topological sort has been completed successfully. All actions have been printed in the output stream.  
else  
    {If KN is not equal to zero, then the relations have violated the hypothesis for a partial order, i.e., the relations contain a loop. We need to print the loop}  
  
    For K in 1.. NA loop  
        QLink[K] <- 0  
    end loop

```

6)   For K in 1 .. NA loop
      P <- Top[K]; Top[K] <- 0
      While P <> 0 and Qlink[Succ(P)] = 0 loop
        Qlink[Succ(P)] <- K
        If P <> 0 then
          P <- Next(P)
        end if
      end loop
    end loop

```

{At this point, QLink[K] will point to one of the predecessors of action K for each action K that has not yet been processed}

```

7)   {Find a K with QLink[K] not = 0. This will be part of the loop.}
      K <- 1
      while (QLink[K] = 0) loop K <- K + 1; end loop

```

```

8)   {Look for loop and mark it.}
      Repeat
        Top[K] <- 1; K <- QLink[K];
      Until Top[K] not = 0

```

```

9)   {Print the loop.}
      While Top[K] not = 0 loop
        Print {process} action K.
        Top[K] <- 0; K <- QLink[K];
      end loop

```

Print the value of K. K is the first node in the loop printed backwards.  
This terminates the failed sort.

# Sample Topological Sort Problems

## **“C” Option:**

Implement the simple version of the topological sort algorithm. Sort the following relations. Process the following relations:  $1 < 2$ ,  $2 < 4$ ,  $2 < 3$ ,  $3 < 10$ ,  $4 < 10$ ,  $7 < 10$ ,  $7 < 6$ ,  $6 < 4$ ,  $5 < 4$ ,  $8 < 5$ ,  $8 < 6$ ,  $1 < 2$ , and  $9 < 8$ . Note that the data contains duplicate relations. your program should not be affected by duplicate data except for additional run time and space.

Now process the following relations:  $1 < 2$ ,  $2 < 4$ ,  $2 < 3$ ,  $3 < 10$ ,  $4 < 10$ ,  $7 < 10$ ,  $7 < 6$ ,  $6 < 4$ ,  $4 < 5$ ,  $5 < 8$ ,  $8 < 6$ ,  $1 < 2$ , and  $9 < 8$ .

## **“B” Option:**

Implement the topological sort algorithm. If a loop is encountered, print the actions that make up the loop. Process the data for the “C” option. You need not implement the “C” Option program.

## **“A” Option:**

Allow the user to specify actions using any programmer defined data type they desire. Process the following data in addition to the “C” option data. A partial specification follows. The “A” option must print the contents of a loop if encountered.

Additional Data: Joe < Mary, Joe < Tom, Mary < Tom, Tom < Bob,  
Sara < Julie, Larry < Bob, Julie < Sam, Mary < Sam,  
and Julie < Larry.

Additional Data: Joe < Mary, Joe < Tom, Tom < Bob, Bob < Mary, ,  
Mary < Tom.

$9 < 2$ ,  $3 < 7$ ,  $7 < 5$ ,  $5 < 8$ ,  $8 < 6$ ,  $4 < 6$ ,  $1 < 3$ ,  $7 < 4$ ,  $9 < 5$ ,  $2 < 8$

$1 < 3$ ,  $3 < 7$ ,  $1 < 3$

Hint: See the generic circular queue page 62-64, I/O definitions page 85-86, and the sample for passing I/O routines to a generic package.

```
generic -- You may modify this as required but observe the spirit.
    type ActionType is private;
    with procedure get(Action: out ActionType);
    with put(Action: in ActionType);

package GenericTopologicalSort is
    TopologicalSort;
end GenericTopologicalSort;

package body GenericTopologicalSort is
    -- This should read (get) the relations and print (put) the results.
    type Node;
    type NodePointer is access Node;
    type Node is record
        Suc: Integer;
        Next: NodePointer;
    end record;

    type SortElement is record
        Count: Integer;
        Top: NodePointer;
    end record;

    SortStructure: Array(ActionType) of SortElement;
    -- other declarations
    procedure TopologicalSort is
    begin -- program to sort and print results; end TopologicalSort;
end GenericTopologicalSort;

with GenericTopologicalSort;
procedure Main is

    type NameType is (Mary, Joe, Tom, Bob, Sara, Julie, Larry, Sam);

    package NameTypeIO is new Ada.Text_IO Enumeration_IO(NameType);
    use NameTypeIO;

    -- Overload definitions for single parameter "get(Action : out ActionType)"
    -- and "put(Action: in ActionType)" for NameTypeIO.

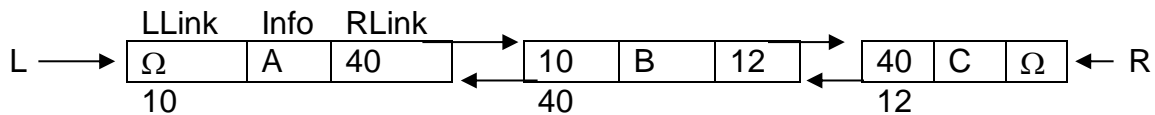
    package NameTopologicalSort is new
        GenericTopologicalSort(NameType, get, put);
    use NameTopologicalSort;
begin
    -- rest of program
end Main;
```



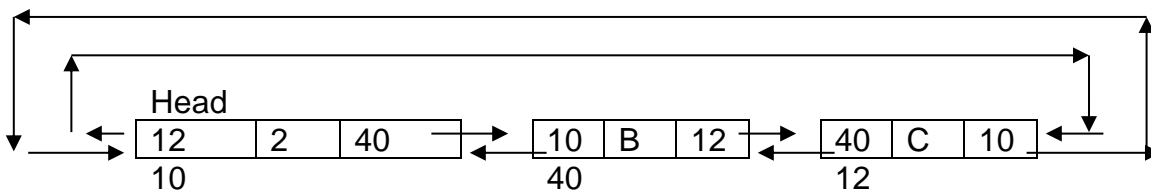
## Doubly linked list.

Doubly linked list makes traversal easy in both directions. They also make the algorithms for random insertion and deletion easier. The penalty of course is the increased space required for the links (double). If the node is large relative to the space required for the links, the additional space for the links may be negligible.

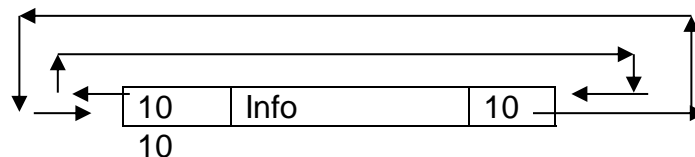
Note that  $L \leftarrow 10$  and  $R \leftarrow 12$ , where L and R point to the left and right end of the list:



**List Head** (using the info field of head to track the number of nodes in the list):



**Empty list with list head:**  $\text{Head.Link} = \text{Head.Rlink} = \text{Head}$  :

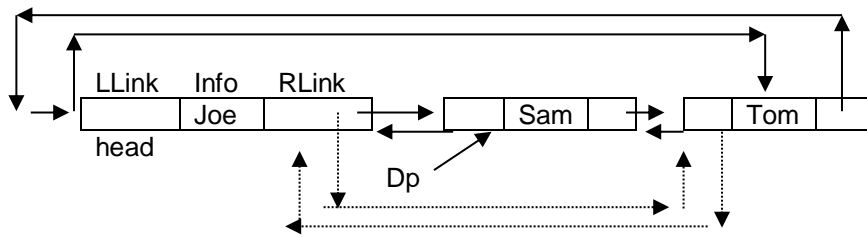


$X.\text{Link} = X.\text{Rlink} = X$

Or

**$\text{Llink}(X) = \text{Rlink}(X) = X$  for any list with one node.** Head or  $X = 10$  in the above example, symmetry is consistent throughout the list.

### Random deletion of node pointed to by Dp:

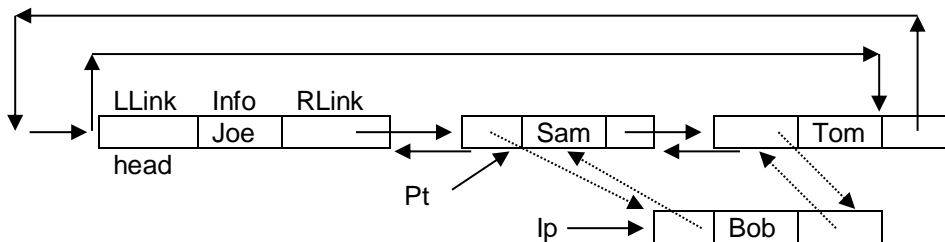


- 1)  $\text{RLink}(\text{LLink}(\text{Dp})) \leftarrow \text{RLink}(\text{Dp})$
- 2)  $\text{LLink}(\text{RLink}(\text{Dp})) \leftarrow \text{LLink}(\text{Dp})$
- 3)  $Y \leftarrow \text{Info}(\text{Dp})$
- 4)  $\text{Dp} \Rightarrow \text{Avail}$

or

- 1)  $(\text{Dp}.\text{Llink}).\text{Rlink} \leftarrow \text{Dp}.\text{Rlink}$
- 2)  $(\text{Dp}.\text{Rlink}).\text{Llink} \leftarrow \text{Dp}.\text{Llink}$
- 3)  $Y \leftarrow \text{Dp}.\text{Info}$
- 4)  $\text{Dp} \Rightarrow \text{Avail}$

### Random insertion of the node at Ip to the right of Pt:

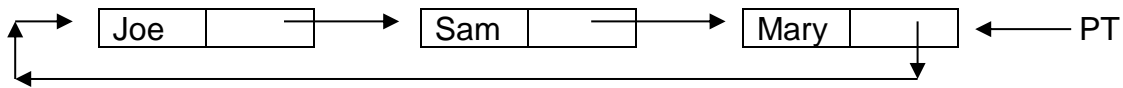


- 1)  $s(\text{Ip}) \leftarrow \text{Pt}$
- 2)  $\text{RLink}(\text{Ip}) \leftarrow \text{RLink}(\text{Pt})$
- 3)  $\text{LLink}(\text{RLink}(\text{Pt})) \leftarrow \text{Ip}$
- 4)  $\text{RLink}(\text{Pt}) \leftarrow \text{Ip}$

or

- 1)  $\text{Ip}.\text{LLink} \leftarrow \text{Pt}$
- 2)  $\text{Ip}.\text{RLink} \leftarrow \text{Pt}.\text{RLink}$
- 3)  $(\text{Pt}.\text{RLink}).\text{LLink} \leftarrow \text{Ip}$
- 4)  $\text{Pt}.\text{RLink} \leftarrow \text{Ip}$

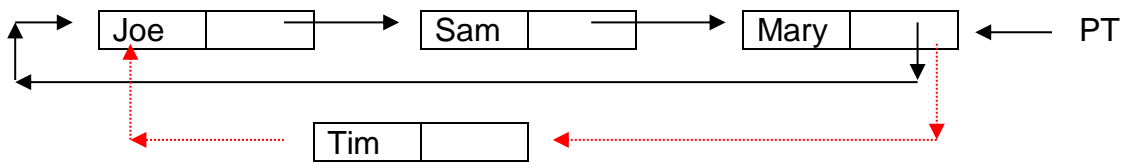
# Link Allocated Circular List



## A) Insert y at the left: e.g., insert Tim.

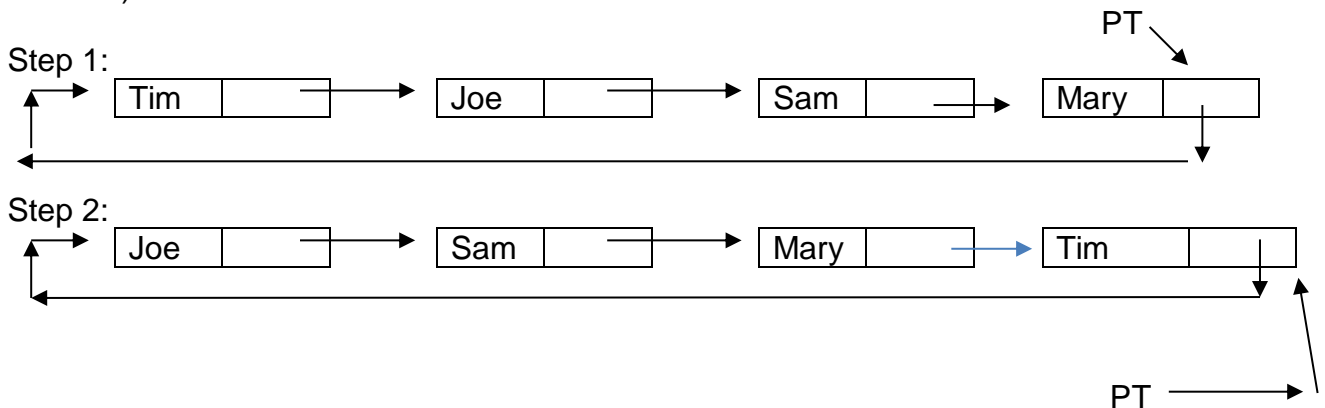
```

P <= Avail
P.Info <- y
if Pt = Ω then {empty?}
    Pt <- P.Link <- P
else
    P.Link <- Pt.Link
    Pt.Link <- P
end if
  
```



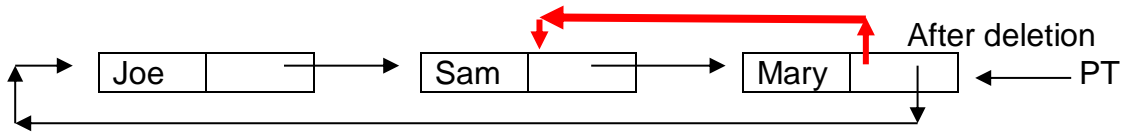
## B) Insert at right: e.g., insert Tim

- 1) Insert at the left
- 2) Pt <- P



Motivations:

- 1) Insert left to process invoices/applicants from the most recent to the oldest.
- 2) Insert right to process invoices/applicants from oldest to newest.

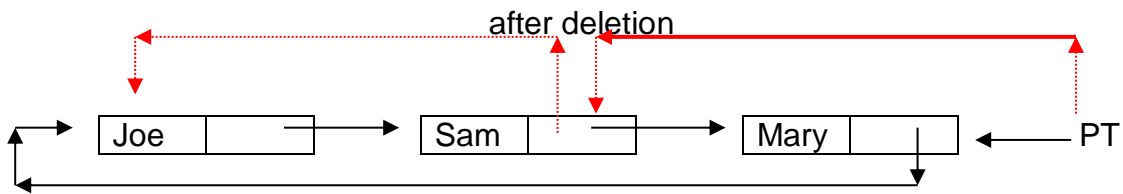


**C) Set y to the contents of the left node and delete the left node: e.g., delete Joe.**

```

if Pt =  $\Omega$  then
    report underflow
else
    P <- Pt.Link
    y <- P.Info
    Pt.Link <- P.Link
    P => Avail
    if P = Pt then {deleted the last node?}
        Pt <-  $\Omega$ 
    end if
end if

```



**D) Delete on the right: e.g., delete Mary.**

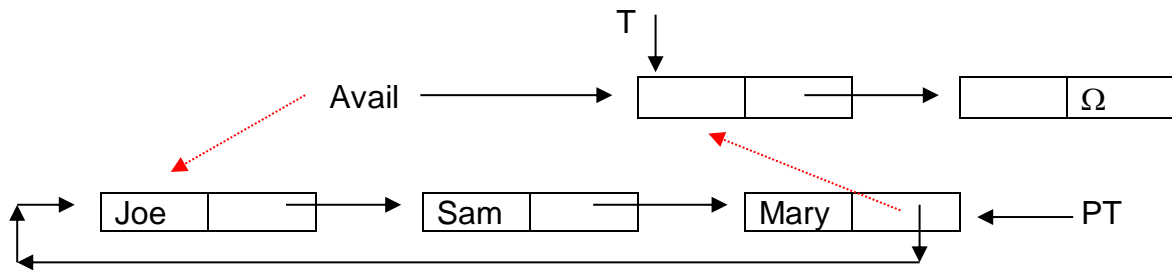
```

if Pt =  $\Omega$  then
    report underflow
else
    {first find the tail}
    P <- Pt
    while P.Link <> Pt loop
        P <- P.Link
    end loop
    {manipulate}
    Dp <- Pt; y <- Dp.Info
    {delete}
    P.Link <- Pt.Link  Pt <- P
    if Dp = Pt then {deleted last node}
        Pt <-  $\Omega$ 
    end if
    Dp => Avail
end if

```

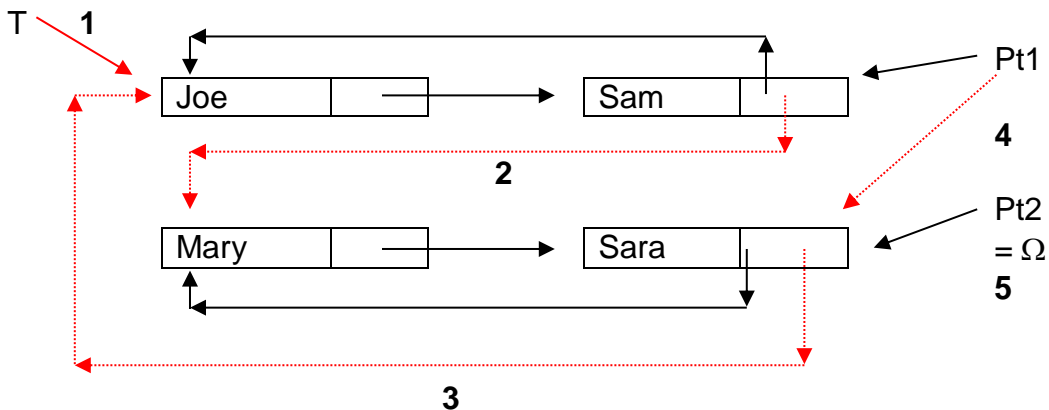
**Return an entire singly linked list pointed to by Pt to the Avail storage list.**

```
{Exchange pointers, i.e., Avail <--> Pt.Link}
T <- Avail
Avail <- Pt.Link
Pt.Link <- T
Pt <- Ω
```

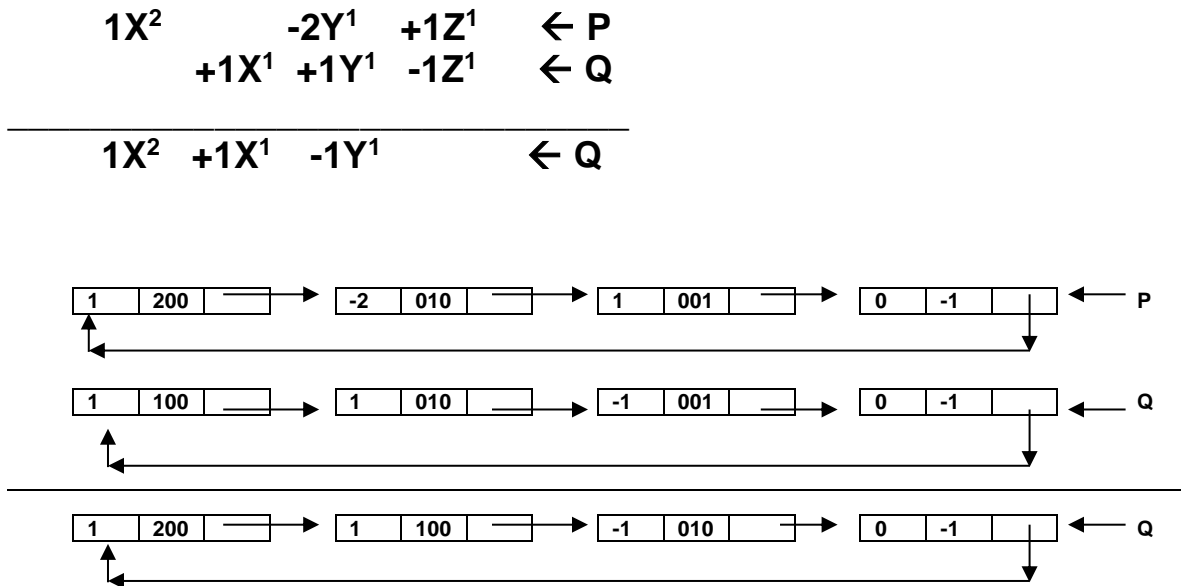


**Concatenate to right:**

- 1) T <- Pt1.Link
- 2) Pt1.Link <- Pt2.Link
- 3) Pt2.Link <- T
- 4) Pt1 <- Pt2
- 5) Pt2 <- Ω



Assume Pt1 and Pt2 point to passenger list for two aircraft scheduled to depart IAH at the same time for Dallas. Neither plane has enough passengers to make it economical. We desire to combine the passengers onto a single flight.



### Polynomial Addition:

This algorithm adds a polynomial pointed to by P to a polynomial pointed to by Q with Q pointing to the sum ( $Q := P + Q$ ). The polynomial pointed to by P is not modified. This algorithm is a modification of an algorithm reported by Knuth in “Fundamental Algorithms” published by Addison-Wesley, pp. 272-274. Each term is of the form coefficient \*  $X^A Y^B Z^C$ . The algorithm assumes positive integer exponents and that each polynomial is terminated with a special sentinel node with the ABC field set to -1. The terms of the polynomials must be in descending order on the ABC field. For convenience, each polynomial is terminated by a sentinel node with a coefficient of 0 and exponent of -1.

#### Assume:

with UnChecked\_Deallocation; {generic procedure to reclaim dynamically allocated memory}

**type Real is digits 7;**

**type Term;**

**type TermPt is access Term;**

{Create procedure to free dynamically allocated memory and place on available storage list for latter use.}

**procedure Free is new UnChecked\_Deallocation(Term, TermPt);**

**type Term is record**

**Coeff: Real;**

**ABC: Integer range -1..999;**

**Link: TermPt;**

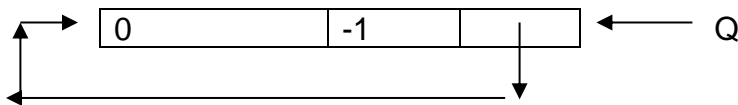
**end record;**

**Algorithm to add two polynomials pointed to by P and Q. The sum is placed in Q.**

Node Format for coefficient \*  $X^A Y^B Z^C$  is

|             |     |      |
|-------------|-----|------|
| coefficient | ABC | Link |
|-------------|-----|------|

The empty polynomial is represented by:



**Start the algorithm:**

P := P.Link; Q1 := Q; Q := Q.Link;

Loop

while (P.ABC < Q.ABC) loop {find next term in P to add to Q}

Q1 := Q; Q := Q.Link;

end loop;

if (P.ABC = Q.ABC) then {terms with equal exponents, add coefficients}

exit when (P.ABC < 0); {Exit the loop, all terms have been added, stop.}

Q.Coef := Q.Coef + P.Coef;

If Q.Coef = 0 then {delete the zero term, i.e., 0 \* any = 0}

Q2 := Q; Q := Q.Link; Q1.Link := Q;

Free(Q2); {Prevent memory hemorrhaging, Q2 => Avail}

P := P.Link; {move to next term in P to add to Q}

else

Q1 := Q; Q := Q.Link; P := P.Link; {moves to next term}

end if;

else

if (P.ABC > Q.ABC) then {Add a new term from P to Q}

{P contains a term that does not exist in Q that must be added.}

Q2 := new Term;

Q2.Coef := P.Coef; Q2.ABC := P.ABC; {copied, now link}

Q2.Link := Q; Q1.Link := Q2; {linked, now move to next term}

Q1 := Q2; P := P.Link;

end if;

end if;

end loop;

## Multiplication of Polynomials:

Note that multiplication is just a series of additions. This algorithm replaces a polynomial pointed to by Q by the sum of polynomial Q and the product of the polynomial pointed to by M times the polynomial pointed to by P. This implies:  $Q := Q + (M * P)$ . Q must initially be set to the empty polynomial defined above.

```

TraverseM: loop
    M := M.Link;
    exit when (M.ABC < 0); {The multiplication has been accomplished, stop.}

    TraverseP:
        loop
            Q1 := Q;
            P := P.Link;
            exit TraverseP when P.ABC < 0;
            T := new Term;
            T.Coef := P.Coef * M.Coef;
            T.ABC := P.ABC + M.ABC;

            Repeat {place in descending order of ABC}
                Q2 := Q1;
                Q1 := Q1.Link;
            until T.ABC >= Q1.ABC;

            if T.ABC = Q1.ABC then {Like term exists.}
                Q1.Coef := T.Coef + Q1.Coef;
                if Q1.Coef = 0 then {Delete 0 Coef term.}
                    Q2.Link := Q1.Link;
                    Free(Q1);
                end if;
                Free(T);
            else if T.ABC > Q1.ABC then {New term.}
                T.Link := Q1;
                Q2.Link := T;
            end if;
        end if;
    end loop TraverseP;
end loop TraverseM;
```

**Try:**  $(X^4 + 2X^3Y + 3X^2Y^2 + 4XY^3 + 5Y^4) * (X^2 - 2XY + Y^2) = X^6 - 6XY^5 + 5Y^6$

polyarit.doc



## Indexing Sequentially Allocated Structures - Arrays

Assume that a multidimensional array is stored in sequential memory locations starting at location  $a_0$ . Assume further that the elements of the array are stored in “**lexicographic order**” of the indices (left to right), sometimes called “row major order.”

**Example: Tradition: plane row col**

**A: array(0..1, 0..1, 0..2) of Anything;**

--We will assume that “anythings” occupy 1 unit of storage.

The size of each dimension (left to right) is  $d_1 := 2$ ,  $d_2 := 2$ , and  $d_3 := 3$ . Hence the required space is  $d_1 * d_2 * d_3 = 2 * 2 * 3 = 12$  units ( one unit per item was assumed).

Location

|     |          |
|-----|----------|
| 100 | A(0,0,0) |
| 101 | A(0,0,1) |
| 102 | A(0,0,2) |
| 103 | A(0,1,0) |
| 104 | A(0,1,1) |
| 105 | A(0,1,2) |
| 106 | A(1,0,0) |
| 107 | A(1,0,1) |
| 108 | A(1,0,2) |
| 109 | A(1,1,0) |
| 110 | A(1,1,1) |
| 111 | A(1,1,2) |

Calculate right to left:

$a_n := 1$

$a_{n-1} := d_n$

$a_k := \text{product all } d_i, \text{ for } k < i \leq n$

$a_0 := \text{base address}$

$a_3 := 1$

$a_2 := d_3 = 3$

$a_1 := d_2 * d_3 = 2 * 3 = 6$

$a_0 := 100$

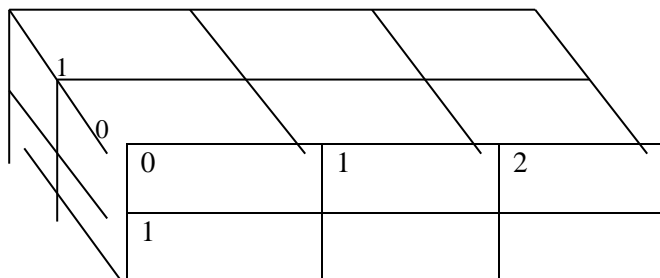
**Formula assuming zero base indexing:**  $\text{Loc}[A(I, J, K)] := \text{base} + \text{offset}$

$\text{Loc}[A(I, J, K)] := a_0 + a_1 I + a_2 J + a_3 K := 100 + 6 * I + 3 * J + 1 * K$

Ex:  $\text{Loc}[A(1, 0, 2)] := 100 + 6 * 1 + 3 * 0 + 1 * 2 = \mathbf{108}$

If there are “C” units of storage for an “anything,” the formula becomes:

$\text{Loc}[A(I, J, K)] := a_0 + (a_1 I + a_2 J + a_3 K) * C$



$d_1 \Rightarrow$  planes,  $d_2 \Rightarrow$  rows,  $d_3 \Rightarrow$  columns

**Example:****A: array(-1..0, 0..1, 1..3) of Anything;**

--We will assume that "anythings" occupy 1 unit of storage.

The size of each dimension is  $d_1 := 2$ ,  $d_2 := 2$ , and  $d_3 := 3$ . Hence the required space is  $d_1 * d_2 * d_3 = 2 * 2 * 3 = 12$  units ( one unit per item was assumed).

Location

|     |             |
|-----|-------------|
| 100 | A(-1, 0, 1) |
| 101 | A(-1, 0, 2) |
| 102 | A(-1, 0, 3) |
| 103 | A(-1, 1, 1) |
| 104 | A(-1, 1, 2) |
| 105 | A(-1, 1, 3) |
| 106 | A(0, 0, 1)  |
| 107 | A(0, 0, 2)  |
| 108 | A(0, 0, 3)  |
| 109 | A(0, 1, 1)  |
| 110 | A(0, 1, 2)  |
| 111 | A(0, 1, 3)  |

Calculate right to left:

 $a_n := 1$  $a_{n-1} := d_n$  $a_k := \text{product all } d_i, \text{ for } k < i \leq n$  $a_0 := \text{base address}$  $a_3 := 1$  $a_2 := d_3 = 3$  $a_1 := d_2 * d_3 = 2 * 3 = 6$  $a_0 := 100$ **Formula: Must convert to zero base indexing (translate the axis):**

$$\text{Loc}[A(I, J, K)] := a_0 + a_1(I+1) + a_2J + a_3(K-1) := 100 + 6*(I+1) + 3*J + 1*(K-1)$$

Ex:  $\text{Loc}[A(-1, 1, 2)] := 100 + 6*(-1 + 1) + 3*1 + 1*(2-1)$   
 $:= 100 + 6*0 + 3 + 1*1 := 100 + 0 + 3 + 1 := 104$

\*\*\*\*\*

**General Case assuming lexicographic order and "c" units of storage per entry.**

$$\begin{aligned} \text{Loc}[A(l_1, l_2, \dots, l_k)] &:= \text{Loc}[A(0, 0, \dots, 0)] + c(d_2 + 1)^{\dots} (d_k + 1)l_1 + \dots \\ &\quad + c(d_k + 1)l_{k-1} + cl_k \\ &:= \text{Loc}[A(0, 0, \dots, 0)] + \sum_{1 \leq r \leq k} a_r l_r \end{aligned}$$

where  $a_r := c \prod_{r < s \leq k} (d_s + 1)$ .

\*\*\*\*\*

Most languages allowing programmer defined enumeration types number the elements of the type from left to right starting with 0.

type Color is (red, green, blue); => red = 0, green = 1, and blue = 2.

type color = (red, yellow, green, blue, purple);

A: array(-10..10, 20..30, red..green, 0..100, red..purple, -10..-5) of anything;

A: array(-10..10, 20..30, red..green, 0..100, Color, -10..-5) of anything;

### Example Assuming Column Major Order as in Fortran:

#### A: array(1..2, 1..2, 1..3) of Anything;

--We will assume that "anythings" occupy 1 unit of storage.

The size of each dimension is  $d_1 := 2$ ,  $d_2 := 2$ , and  $d_3 := 3$ . Hence the required space is  $d_1 * d_2 * d_3 = 2 * 2 * 3 = 12$  units ( one unit per item was assumed).

Location

|     |            |
|-----|------------|
| 100 | A(1, 1, 1) |
| 101 | A(2, 1, 1) |
| 102 | A(1, 2, 1) |
| 103 | A(2, 2, 1) |
| 104 | A(1, 1, 2) |
| 105 | A(2, 1, 2) |
| 106 | A(1, 2, 2) |
| 107 | A(2, 2, 2) |
| 108 | A(1, 1, 3) |
| 109 | A(2, 1, 3) |
| 110 | A(1, 2, 3) |
| 111 | A(2, 2, 3) |

Calculate left to right (opposite lexicographic order):

$a_0 := 100$

$a_1 := 1$

$a_2 := d_1 = 2$

$a_3 := d_1 * d_2 = 2 * 2 = 4$

$a_0 := \text{base address}$

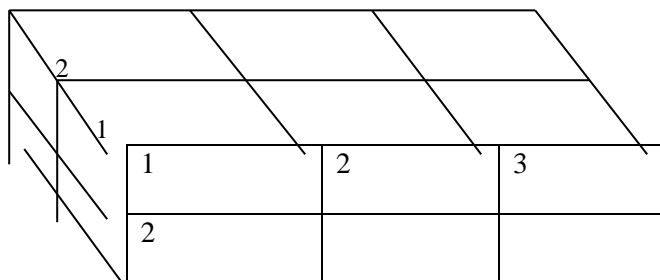
$a_1 := 1$

$a_i := \text{product all } d_k, \text{ for } k \geq 1 \text{ \& } k < i$

### Formula assuming non-zero base indexing:

$\text{Loc}[A(I, J, K)] := a_0 + a_1(I-1) + a_2(J-1) + a_3(K-1) := 100 + 1*(I-1) + 2*(J-1) + 4*(K-1)$

Ex:  $\text{Loc}[A(1,2,3)] := 100 + 1*(1-1) + 2*(2-1) + 4*(3-1)$   
 $:= 100 + 1*0 + 2*1 + 4*2 := 100 + 0 + 2 + 8 := 110$



$d_1 \Rightarrow \text{planes}, d_2 \Rightarrow \text{rows}, d_3 \Rightarrow \text{columns}$

Addressing scheme for upper triangular matrix:

Note we only need to store the entries  $A(j,k)$  for  $0 \leq k \leq j \leq n$ :

|        |        |     |        |
|--------|--------|-----|--------|
| A(0,0) |        |     |        |
| A(1,0) | A(1,1) |     |        |
| ooo    |        |     |        |
| A(n,0) | A(n,1) | ooo | A(n,n) |

Assume we know that all other entries are zero, or that  $A(j,k) = A(k,,j)$ . We wish to store only the lower triangular matrix in consecutive memory locations. This implies a function of the form:

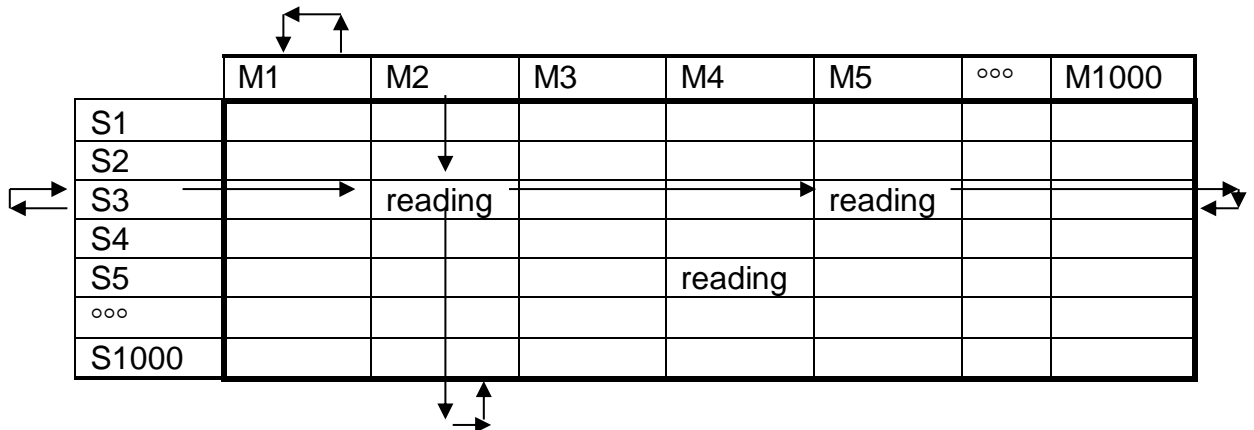
$$\text{Loc}[A(J,K)] = a_0 + f_1(J) + f_2(K).$$

Assuming lexicographic order of the indices, this can be solved if  $f_2 := K$  and  $f_1 := (J(J+1))/2$  for  $c = 1$ .

Hence:  $\text{Loc}[A(J, K)] := \text{Loc}[A(0,0)] + (J(J+1))/2 + K$ . See Knuth (Fundamental Algorithms, p297) for a far better way to store triangular matrices if we are fortunate enough to have two with the same dimensions.

Hint:  $1 + 2 + \dots + N = \sum_{I=1}^N I = \frac{N(N+1)}{2}$ .

Assume a 1000 by 1000 array. Each row could represent a weather station and the corresponding column the current measurements. At any point in time 800 stations will be queried for information to drive our weather simulation. We have insufficient compute power to utilize more measurements (even though more measurements are desirable). It will require  $1 \times 10^6$  entries for space. A better solution requiring  $800 \cdot H + 2000$  (where  $H$  is the space for a reading as defined below) entries would be:



Readings node format:

|         |     |        |       |       |
|---------|-----|--------|-------|-------|
| reading | row | column | rowPt | colPt |
|---------|-----|--------|-------|-------|

# Trees

A **tree** is a finite set  $T$  of one or more nodes which satisfy the following conditions: 1) There is one specially designated node called the root of the tree,  $T.root$ ; and 2) The remaining nodes (excluding the root) are partitioned into  $m \geq 0$  disjoint sets  $T_1, \dots, T_m$ . Each of these sets in turn is a tree. The tree  $T_1, \dots, T_m$  are called the **subtrees** of the root.

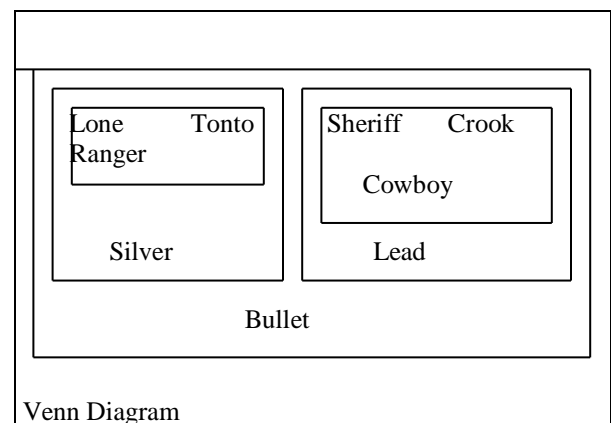
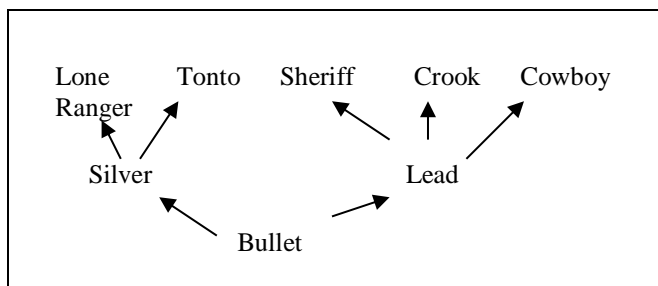
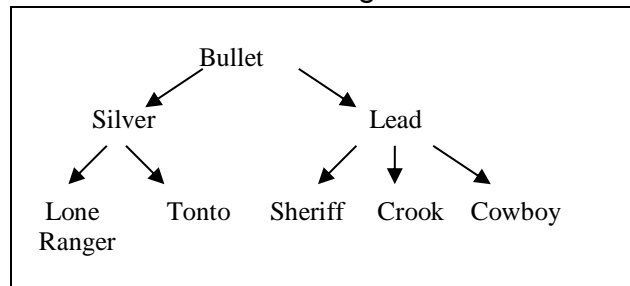
The number of subtrees of a node is called the **degree** of the node.

A node of degree zero is called a **terminal** node or a **leaf**.

The **level** of a node with respect to  $T$  is defined as: 1) The root is at level zero (0); 2) Other nodes have a level that is one higher than they have with respect to the subtree  $T_j$ , of the root which contains them. We refer to father, grandfather, son, brother, etceteras for terminology.

A **binary tree** is a finite set of nodes which is either empty, or consists of a root and two disjoint binary trees called the left subtree and the right subtrees of the root.

01 Bullet  
 02 Silver  
   10 Lone Ranger  
   10 Tonto  
 02 Lead  
   10 Sheriff  
   10 Crook  
   10 Cowboy



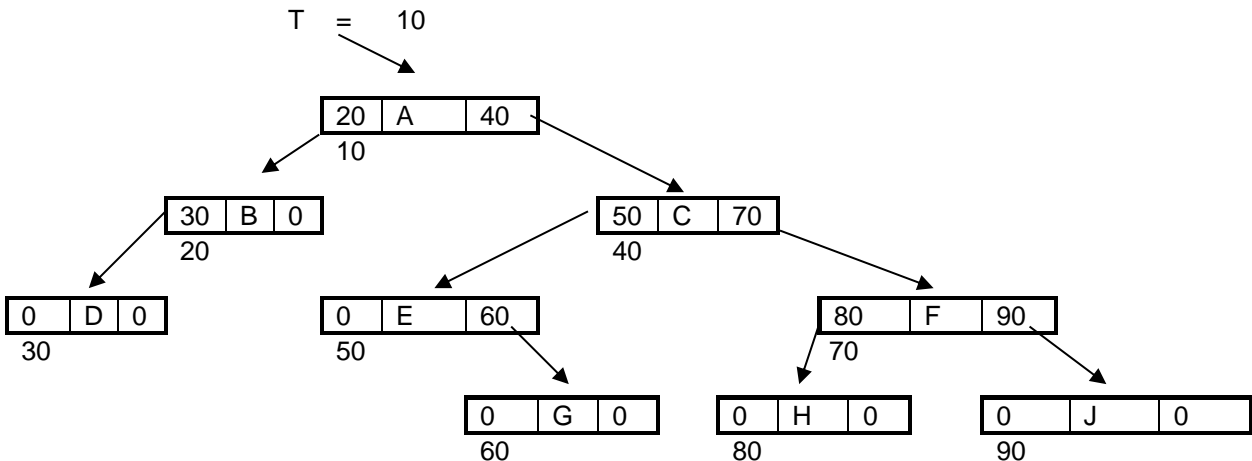
Trees are typically used to represent relationships in structured data types (non-scaler) by language translator writers.

|                                                                                                                                                                                                                                                                                                                                      |                                                                                                                                                                                                   |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> type Silver is record     LoneRanger:---;     Tonto: ---; end record;  type Lead is record     sheriff: --;     crook: --;     cowboy: --; end record;  type Bullet is record     Hero: Silver;     Lesser: Lead; end record; </pre>                                                                                           | <pre> struct Silver {     int LoneRanger;     int Tonto; }  struct Lead {     int sheriff;     int crook;     int cowboy; end record;  struct Bullet {     Silver hero;     Lead lesser; } </pre> |
| <pre> class Silver {     // data structures     int LoneRanger;     int Tonto;     // methods     shootsStraight( target ); }  class Lead {     //data structures     int sheriff;     int crook;     int cowboy;     // methods }  class Bullet {     // data structures      Silver hero;     Lead lesser;     // methods } </pre> |                                                                                                                                                                                                   |

If the relative order of the subtrees  $T_1, \dots, T_m$ , where  $m \geq 2$ , is important, then we say the tree is **ordered**.

If we do not wish to consider two trees as different when they differ only in the orientation of the subtrees, then the trees are said to be **oriented** as opposed to ordered.

A **forest** is a set of zero or more disjoint trees. A forest may be converted to a tree by adding a common root.



Recursive definitions for traversing a binary tree:

### Preorder Traversal

- 1) Visit the root
- 2) Traverse the left subtree
- 3) Traverse the right subtree

A, B, D, C, E, G, F, H, J

### Inorder Traversal

- 1) Traverse the left subtree
- 2) Visit the root
- 3) Traverse the right subtree

D, B, A, E, G, C, H, F, J

### Postorder Traversal

- 1) Traverse the left subtree
- 2) Traverse the right subtree
- 3) Visit the root

D, B, G, E, H, J, F, C, A

### Traverse a Binary Tree in Inorder:

This algorithm traverses a binary tree pointed to by T in inorder using an auxiliary stack A.

```
1) Set the stack A empty, and  $P \leftarrow T$ .
2) loop
    if  $P \neq \text{null}$  then;
         $P \Rightarrow A$  // Push A (location) into stack A
         $P \leftarrow P.\text{LLink}$ ;
    else
        exit when A is empty; {the algorithm terminates}
         $P \Leftarrow A$ ; // pop stack A
        Visit P.Node;
         $P \leftarrow P.\text{RLink}$ ;
    end if
end loop;
```

Note: The notation  $P \Rightarrow A$  means to push the value P into stack A. Conversely,  $P \Leftarrow A$  means to pop stack A and assign the value popped to P.

**Stack Space Required:** approximately  $\log_2 N$ , where N is the number of items in a reasonably well balanced tree.

### Traverse a Binary Tree in PreOrder:

This algorithm traverses a binary tree pointed to by T in PreOrder using an auxiliary stack A.

```
1) Set stack A empty and  $P_t \leftarrow T$ 
2) loop
    if  $P_t \neq \text{null}$  then
        visit  $P_t.\text{node}$ 
         $P_t \Rightarrow A$ 
         $P_t \leftarrow P_t.\text{Llink}$ 
    Else
        Exit when A is empty
         $P_t \Leftarrow A$ 
         $P_t \leftarrow P_t.\text{Rlink}$ 
    End if
end loop
```



with Ada.Text\_IO; use Ada.Text\_IO; -- data use : H, K, A, B, L

```
procedure alptre1 is  
  type Node;  
  type Tree is access Node;  
  type Node is record  
    Value: Character;  
    Left, Right: Tree;  
end record;
```

Ch: character;

**Root: tree;**

```
procedure Insert(t: in out Tree; v: Character) is -- build the tree  
begin  
  if t = null then  
    t := new Node; t.Value := v;  
    t.Left := null; -- actually set to null automatically by Ada  
    t.Right := null;  
  else  
    if v < t.Value then  
      Insert(t.Left,v);  
    else  
      Insert(t.Right,v);  
    end if;  
  end if;  
end Insert;
```

```
procedure Inorder(t:Tree) is -- recursive inorder tree traversal  
begin  
  if t /= null then  
    Inorder(t.Left); put("tree sort "); -- 1 Traverse Left  
    put(t.Value); new_line; -- 2 Visit  
    inorder(t.Right); -- 3 Traverse Right  
  end if;  
end Inorder;
```

```
begin -- read info and place in sorted order recursively  
  root := null; -- All pointers set to null initially automatically in ada  
  loop  
    put("Enter a character, 'Z' to exit: "); get(Ch); exit when Ch = 'Z';  
    Insert(Root, Ch);  
  end loop;  
  
  Inorder(root); -- print in sorted order  
  
end;
```

```

with Ada.Text_IO; use Ada.Text_IO;
procedure alphatre2 is      -- Builds tree recursively.
  type Node;                -- Traverses iteratively.
  type Tree is access Node;
  type Node is record
    Value: Character; Left, Right: Tree;
  end record;

  Ch: character;
  Root: tree;

  procedure Insert(t: in out Tree; v: Character) is -- build the tree
  begin
    if t = null then
      t := new Node; t.Value := v;
      t.Left := null; t.Right := null; -- Actually the default in Ada is null when allocated.
    else
      if v < t.Value then
        Insert(t.Left,v);
      else
        Insert(t.Right,v);
      end if;
    end if;
  end Insert;

  procedure Inorder(t:Tree) is -- Iterative inorder tree traversal.
  Stack: array(1..10) of Tree; -- Balance tree requires log base 2
  Knt: integer;                -- of N space, N is number nodes.
  Pt: Tree;
  begin
    Knt := 0; Pt := t;-- Set stack empty, Pt to the root of the tree.
    loop
      if Pt /= null then
        Knt := Knt + 1; Stack(Knt) := Pt; Pt := Pt.Left;
      else
        exit when Knt = 0; -- Traversed whole tree.
        Pt := Stack(Knt); Knt := Knt - 1;
        put("tree sort "); put(Pt.Value); new_line; Pt := Pt.Right;
      end if;
    end loop;
  end Inorder;

begin -- read info and place in sorted order recursively
  root := null; -- All pointers set to null initially automatically in ada
  loop
    put("Enter a character, 'Z' to exit: "); get(Ch); exit when Ch = 'Z';
    Insert(Root, Ch);
  end loop;

  Inorder(root); -- print in sorted order
end;

```

```

procedure Inorder(t:Tree) is -- Iterative inorder tree traversal.
    Stack: array(1..10) of Tree; -- Balance tree requires log base 2
    Knt: integer;                -- of N space, N is number nodes.
    Pt: Tree;
begin
    Knt := 0;
    Pt := t;-- Set stack empty, Pt to the root of the tree.
    loop
        if Pt /= null then
            Knt := Knt + 1;  Stack(Knt) := Pt;  -- push location in stack.
            Pt := Pt.Left;
        else
            exit when Knt = 0; -- Traveresed whole tree.
            Pt := Stack(Knt);  Knt := Knt - 1;
            put("tree sort "); put(Pt.Value); new_line;
            Pt := Pt.Right;
        end if;
    end loop;
end Inorder;

```

## Implementation Examples

For additional examples see:

|  | Problem | Document                 | Page    |  |
|--|---------|--------------------------|---------|--|
|  | C++     | Data Structures Programs | 82 - 88 |  |
|  | Java    | Data Structures Programs | 21 - 22 |  |
|  |         | Data Structures Programs |         |  |

### Algorithm “Recursive Search Tree Insert:”

This algorithm builds a binary search tree (ascending key order) given “key” is the key to be inserted, “t” a pointer to a tree node, and the empty tree is represented by  $\text{Root} \leftarrow \Omega$ . The node format is

|           |             |            |
|-----------|-------------|------------|
| Left link | information | Right link |
|-----------|-------------|------------|

```
insertKey( key: dataType, t: inout treePointer)
begin
    if (t =  $\Omega$ )                                -- insert the new node at location “t.”
        t <= Avail;                                -- we assume storage is available.
        t.leftLink  $\leftarrow$  t.rightLink  $\leftarrow$   $\Omega$ ;
        t.info  $\leftarrow$  key;                        -- new desired information
    else if (key < t.info)                        -- insert in left subtree
        insertKey( key, t.leftLink);
    else  -- insert in right subtree
        insertKey(key, t.rightLink);
    endif;
endif;
end insert;
```

### Traverse a Binary Tree in PostOrder:

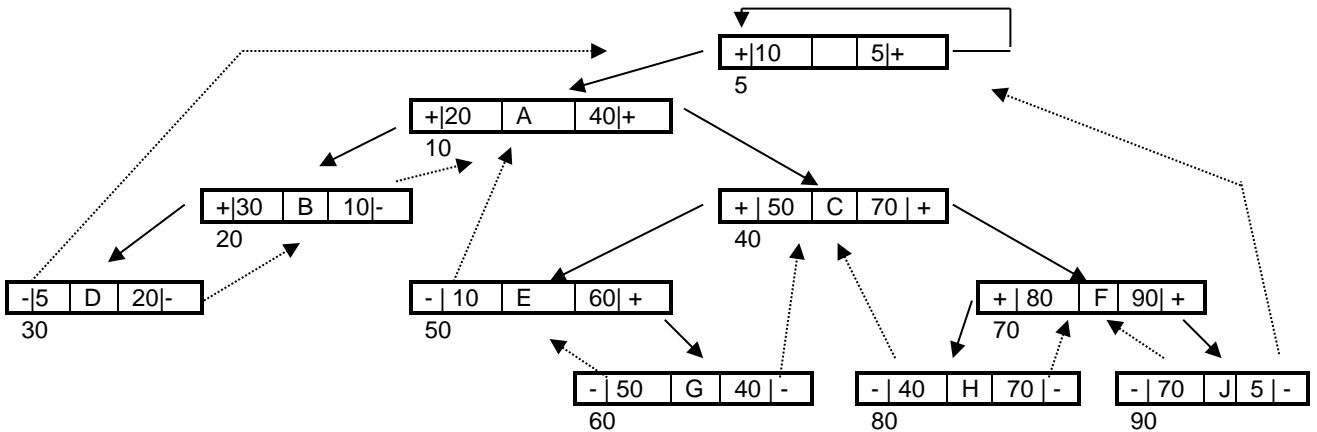
This algorithm traverses a binary tree pointed to by T in postorder using an auxiliary stack A.

```
1) Set the stack A empty, and P := T.
2) loop
    if P <> null then;
        A <= (P, 0);
        P := P.LLink;
    else
        exit algorithm when A is empty; {the algorithm terminates}
        (P, d) <= A;
        if d = 0 then
            A <= (P, 1);
            P := P.Rlink;
        else
            repeat
                Visit P.Node;
                exit algorithm when A is empty; {terminate}
                (P, d) <= A;
                if d = 0 then
                    A <= (P, 1);
                    P := P.Rlink;
                end if
            until d = 0;
        end if;
    end if
end loop;
```

Note: The notation  $A \leq (P, 0)$  means to push the value P and 0 into the same element of stack A. Conversely,  $(P, d) \leq A$  means to pop stack A and assign the values popped to P and d. A "0" is used to indicate traversing a node to the left and 1 to indicate traversal to the right.

**Note that the majority of link fields in this representation are null and contain no additional knowledge. A. J. Perlis and C. Thornton suggested threaded trees as follows:**

|     |                                               |
|-----|-----------------------------------------------|
| P*  | address of successor of P.Node in preorder    |
| P\$ | address of successor of P.Node in inorder     |
| P#  | address of successor of P.Node in postorder   |
| *P  | address of predecessor of P.Node in preorder  |
| \$P | address of predecessor of P.Node in inorder   |
| #P  | address of predecessor of P.Node in postorder |



**P\$: Symmetric Inorder Successor in a threaded binary tree (P\$).**

Assume P points to a node of a threaded binary tree. This algorithm sets  $Q := P\$$ , where  $P\$$  is the inorder successor. The tree is assumed to be threaded in inorder.

**Q := P.RLink; {Look right}**

If P.RTag = "-" then

null; {P points to the inorder successor}

else

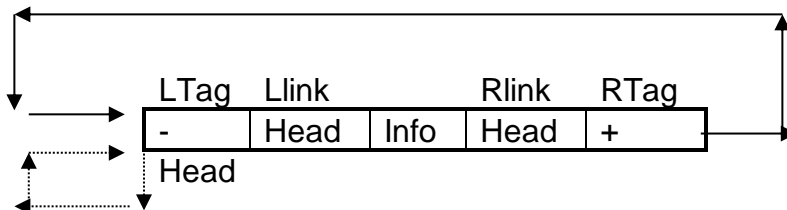
**{Search to left.}**

```
while Q.LTag = "+" loop Q := Q.LLink; end loop;
```

end if;

### Empty Tree:

Assume the tree is pointed to by the left link at the node at location Head. If the left link of Head is not a link, the tree exists; if the left link of Head is a link, the the tree is empty.



**P\*: Symmetric Preorder Successor in a threaded binary tree (P\*).**

Assume P points to a node of a threaded binary tree. This algorithm sets  $Q := P^*$ , where  $P^*$  is the preorder successor. The tree is assumed to be threaded in inorder.

```
if P.LTag = "+" then
    Q := P.LLink;
else
    Q := P;
    while Q.RTag <> "+" loop Q := Q.RLink; end loop;
    Q := Q.RLink;
end if;
```

**\$P: Symmetric Inorder Predecessor of Node P  
(in a threaded binary tree (\$P)).**

Assume P points to a node of a threaded binary tree. This algorithm sets  $Q := \$P$ , where  $\$P$  is the inorder predecessor. The tree is assumed to be threaded in inorder.

```
Q := P.LLink; {Rtag = "-" points to inorder successor of where we start}
if P.LTag <> "-" then
    while Q.RTag = "+" loop Q := Q.RLink; end loop;
end if;
```

**#P: Symmetric Predecessor of Node P in Postorder  
(in a threaded binary tree (#P)).**

Assume P points to a node of a threaded binary tree. This algorithm sets  $Q := P\$$ , where  $P\$$  is the inorder successor. The tree is assumed to be threaded in inorder.

```
Q := P;
if P.RTag = "+" then
    Q := P.RLink;
else
    while Q.LTag <> "+" loop Q := Q.LLink; end loop;
    Q := Q.LLink;
end if;
```

This algorithm attaches a node pointed to by Q as the right subtree of the node pointed to by P if the right subtree is empty, i.e., if P.RTag = "-". It inserts the node pointed to by Q between the node pointed to by P and Node(P.RLink) otherwise. The tree is assumed to be threaded in inorder. The tree must have an appropriate tree head node if empty (page 4).

```
Q.Rlink := P.RLink; Q.RTag := P.RTag;
P.Rlink := Q; P.RTag := "+"; Q.LLink := P; Q.LTag := "-";
```

**{ Q\$: Symmetric Inorder Successor in a threaded binary tree works even though Q\$.LLink now points to P.Node instead of Q.Node. This step is only required if nodes are inserted in the midst of a threaded tree. If all nodes are inserted as leaves, only the first step is required.}**

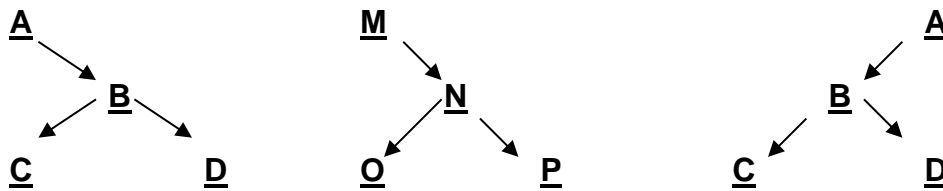
88



Two binary trees T1 and T2 are **similar** (have the same shape) if they have the same structure: a) they are both empty, or b) they are both nonempty and their left and right subtrees are respectively similar. Example: in many companies and the government managers at the same level (pay grade, civil service rating) are considered interchangeable (similar).

Two binary trees T1 and T2 are said to be **equivalent** if they are similar and if corresponding nodes contain the same information. This can be stated more formally as: Let U.Info denote the information contained in the node pointed to by U. Two trees are equivalent if and only if a) they are both empty, or b) they are both nonempty and  $\text{info}(\text{root}(T1)) = \text{info}(\text{root}(T2))$  and their left and right subtrees are similarly equivalent.

Checking for equivalence and similarity can be accomplished in practice for two threaded binary trees by traversing them in preorder and checking the Info and Tag fields.



As an example, in the Federal Civil Service managers with the same rating are supposed to be plug compatible at the same level in different branches, e.g., agriculture, defense, homeland security, etcetera.

## Copy a Binary Tree

Let Head be the address of the list head of a binary tree T (i.e., T is the left subtree of Head; Head.LLink is a pointer to the tree). Let U.Node be a node with an empty left subtree. This algorithm makes a copy of T and the copy becomes the left subtree of U.Node. If U.Node is the list Head of an empty binary tree, then this algorithm makes the empty tree into a copy of T (see Knuth Fundamental Algorithms).

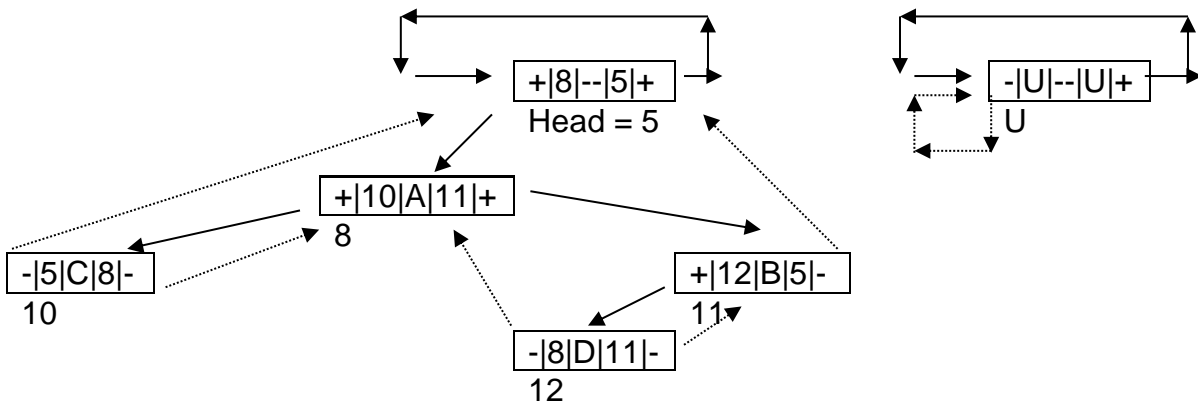
```

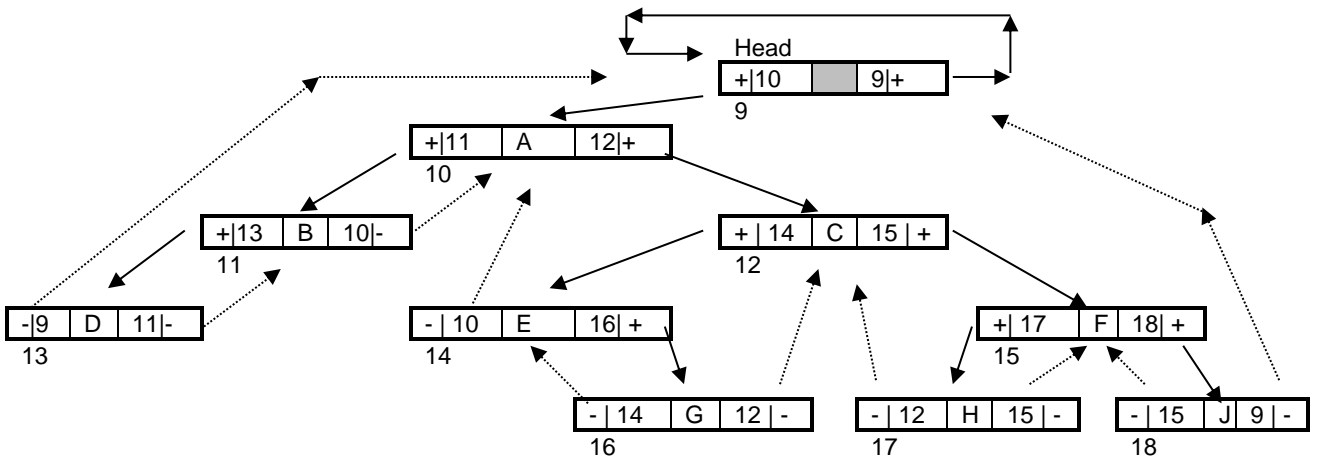
Copy1:      P := Head; Q := U;

Copy2:      loop
              if P.Node has a nonempty left subtree then
                  R <= Avail;
                  Attach R.Node to the left of Q.node;
              end if;
              P := P*;
              Q := Q*;
              exit when P = Head;
              {or Equivalently, if Q = U.Rlink, assuming U.Node
               has a nonempty right subtree}
              if P.Node has a nonempty right subtree then
                  R <= Avail;
                  Attach R.Node to the right of Q.Node;
              end if;

              Q.Info := P.Info; {copy all desired fields}
            end loop;
  
```

For threaded trees, use the insertion algorithm already developed. For nonthreaded trees, a stack is required to find P\* and Q\*.





Assume the tree is built from the top down with no deletions in ascending order by address. Many trees are built in this manner. It is possible to represent a threaded tree in an array without storing the tag fields. If a link field points to a smaller address than our current location, it must be a thread due to the way we have built the tree.

|    |    |      |    |
|----|----|------|----|
| 9  | 10 | Head | 9  |
| 10 | 11 | A    | 12 |
| 11 | 13 | B    | 10 |
| 12 | 14 | C    | 15 |
| 13 | 9  | D    | 11 |
| 14 | 10 | E    | 16 |
| 15 | 17 | F    | 18 |
| 16 | 14 | G    | 12 |
| 17 | 12 | H    | 15 |
| 18 | 15 | J    | 9  |
| 19 |    |      |    |
| 20 |    |      |    |

|    |   |    |      |    |   |
|----|---|----|------|----|---|
| 9  | + | 10 | Head | 9  | + |
| 10 | + | 11 | A    | 12 | + |
| 11 | + | 13 | B    | 10 | - |
| 12 | + | 14 | C    | 15 | + |
| 13 | - | 9  | D    | 11 | - |
| 14 | - | 10 | E    | 16 | + |
| 15 | + | 17 | F    | 18 | + |
| 16 | - | 14 | G    | 12 | - |
| 17 | - | 12 | H    | 15 | - |
| 18 | - | 15 | J    | 9  | - |
| 19 |   |    |      |    |   |
| 20 |   |    |      |    |   |

**type** **TreeNode** is record

    Llink: Integer;   Info: user\_defined;   Rlink: Integer;  
**end record;**

**TreeSpace:** Array(1..N) of **TreeNode**;

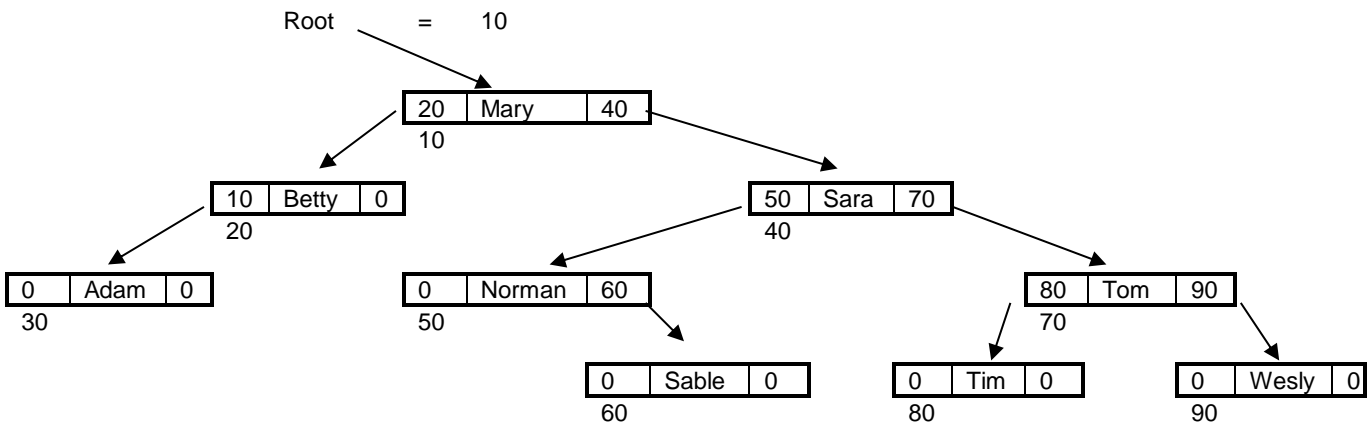
**Avail** : Integer := 1; -- Avail initially <-- 9 in the above diagram.

Hence **PT** <= **AVAIL** implies **Pt** := **PT** + 1;

**Build subtrees** as shown above in order: A, B, C, ...

Tree1.doc

## Binary Search Tree



**Input:** Mary, Betty, Adam, Sara, Norman, Tom, Sable, Wesly, Tim

### Algorithm: Binary Search Tree Search Insertion

Assume a binary search tree pointed to by Root and a key, Akey, we wish to locate or insert if not already in the tree. This algorithm searches the binary search tree for a node containing the value AKey. If such a node is found, the algorithm allows for processing the information in the node. If the key AKey is not found in the tree, AKey is inserted as a new node. The resulting tree is a binary search tree. A node at location P is assumed to contain at least the following fields:

|      |       |     |       |
|------|-------|-----|-------|
| Node | LLink | Key | RLink |
|------|-------|-----|-------|

Empty subtrees are indicated by the appropriate link field set to Null.

type Node;

type NodePointer is access Node;

type Node is

record LLink: NodePointer; Key: KeyType; RLink: NodePointer; end record

### Basic recursive definition to be applied at each node:

- 1) If AKey < key of the node; search to the left if the left subtree exists, else insert Akey as a new left subtree and stop.
- 2) If AKey > key of the node; search to the right if the right subtree exists, else insert Akey as a new right subtree and stop.
- 3) If AKey = key of the node, process the nodes data fields and stop.

a) Expected search probes  $\log_2 N$  if balanced ,b)  $\sim \log_2 N$  all permutations ,c) worst case N, best 1, d) allows binary search, e) inorder traversal produces ascending sorted order, f) insertion requires  $O(\log_2 N)$  operations.

### Iterative Algorithm to insert node in Search Tree:

```
if Root = Null then -- Insert as the root of the tree.
    AllocateNode(Q, AKey);
    Root := Q;
else -- Tree is not empty. Locate a match with existing node or position to insert new node.
    P <- Root;
    Loop -- Search left and right for a match or insert in tree if not found.
        if Akey < P.Key then -- Search to left
            if P.LLink <> Null then
                P <- P. Llink;
            else
                AllocateNode(Q, AKey); -- Insert node as left subtree.
                InsertNode(P, Q, AKey);
                exit loop; -- New node inserted.
            end if;
        elsif Akey > P.Key then -- Search to right.
            if P.RLink <> Null then
                P <- P.RLink;
            else
                AllocateNode(Q, AKey); -- Insert node as right subtree.
                InsertNode(P, Q, AKey);
                exit loop; -- New node inserted.
            end if;
        else -- Implies that Akey matches P.Key.
            -- Node has been found, maipulate the data fields as desired.
            exit loop;
        end if;
    end loop;
end if;
```

```
procedure AllocateNode(Q: out NodePointer; Akey: in KeyType) is
begin -- Allocates and places AKey in node pointed to by Q.
    Q <= Avail; Q.Key <- Akey; Q.LLink <- Q.RLink <- Null;
end;
```

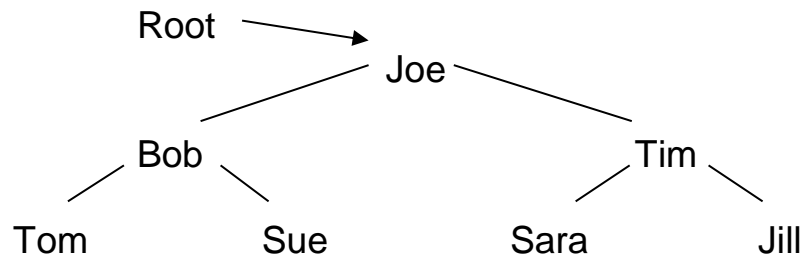
```
procedure InsertNode(P, Q: in out NodePointer; Akey in KeyType) is
begin -- Inserts the Node pointed to by Q as a subtree of P.
    if Akey < P.Key then -- Insert Akey as left subtree of P.
        P.LLink <- Q;
    else -- Insert Akey as right subtree of P.
        P.RLink <- Q;
    end if;
end;
```

## Binary Search Tree Deletion:

This algorithm deletes a random node pointed to by Q from a binary search tree as described above. The resulting tree is a binary search tree. Note that either Q = Root or Q = P.LLink or Q = P.RLink for some node pointed to by P. Q is reset to reflect the deletion.

```
T <- Q;
if T.RLink = Null then
    Q <- T.LLink;
    T => Avail;
else
    if T.LLink = Null then -- Easy delete if LLink = Null
        Q <- T.RLink;
        T => Avail;
    end if;
    -- find successor in inorder
    R <- T.RLink; -- From semetry, step one node to right, search left if required.
    if R.LLink = Null then -- found the inorder successor.
        R.LLink <- T.LLink;
        Q <- R;
        T => Avail;
    else
        S <- R.LLink; -- Search to the left looking for null LLink.
        While S.LLink <> Null loop
            R <- S; S <- R.LLink;
        end loop;
        -- At this point S will be the inorder successor of Q.
        S.LLink <- T.LLink; R.LLink <- S.RLink;
        S.RLink <- T.RLink; Q <- S;
        T => Avail;
    end if;
end if;
if P = Root then
    Root <- Q
Else {We have deleted the first node in the left or right subtree of P.}
    If deleting P.LLink then
        P.LLink <- Q
    Else
        P.RLink <- Q.
end if;
```

**T. N. Hibbard** proved in 1962 that after a random element is deleted from a random binary search tree, by the above algorithm, that the resulting tree is still random. Actual experience (emperical) suggests that over time trees become more balanced hence reducing the length of the search path for random keys.



**In Order: LVR**

Tom, Bob, Sue, Joe, Sara, Tim, Jill

Sort Order: Manager with left and right subtrees properly oriented.

**Pre Order: VLR**

Joe, Bob, Tom, Sue, Tim, Sara, Jill

Sort Order: Manager precedes employee at every level.

**Post Order: LRV**

Tom, Sue, Bob, Sara, Jill, Tim, Joe

Sort Order: Employees precede manager at every level.

\*\* At each level, the nodes are in the same relative order, e.g., leaves: Tom, Sue, Sara, Jill

**No Name 1: RVL**

Jill, Tim, Sara, Joe, Sue, Bob, Tom

Sort Order: Reverse of In Order.

**No Name 2: RLV**

Jill, Sara, Tim, Sue, Tom, Bob, Joe

Sort Order: Reverse of Pre Order.

**No Name 3: VRL**

Joe, Tim, Jill, Sara, Bob, Sue, Tom

Sort Order: Reverse of Post Order

\*\* At each level, the nodes are in the same relative order, e.g., leaves: Jill, Sara, Sue, Tom

For a perfectly balanced tree, the stack space required for an inorder traversal is approximately  $\log_2 N$ .

The worst case space requirement for inorder traversal is  $N$  when all right links are null.

The minimum stack space for inorder traversal is 1 when all left links are null.

The number of possible traversals is:

$$(V|L|R) (V|L|R) (V|L|R) = 3^3 = 27.$$

Note give the preorder and inorder traversal of the tree, the original tree can be reconstructed.

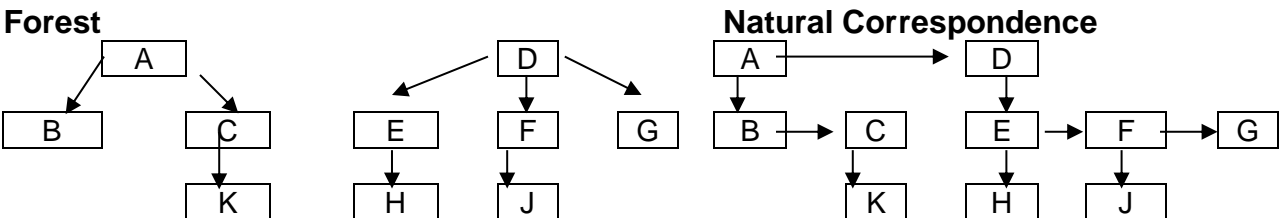
Minimizing the average path length to nodes minimizes the size of the stack required to traverse the tree.



# General M-Ary Trees

-- In file AIEVAL.doc

In general a tree of order **M**, M-ary, may be represented by a corresponding binary tree. This is useful if the desired operations may be accomplished in a more natural fashion.



To create the natural correspondence:

- 1) Connect all sons left to right.
- 2) Remove all vertical links except from a father to the left most son.

| Preorder for a Forest:                                                                                                                              | Postorder for a Forest                                                                                                                                |
|-----------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| a) Visit the root of the first tree;<br>b) traverse the subtrees of the first tree (in preorder);<br>c) traverse the remaining trees (in preorder). | a) Traverse the subtrees of the first tree (in postorder);<br>b) visit the root of the first tree;<br>c) traverse the remaining trees (in postorder). |

ABCKDEHFJG (Enrron pay model)

BKCAHEJFGD (Crysler – Lee Iacocca)

(A(B,C(K)), D(E(H), F(J), G))

((B, (K) C) A, ((H) E, (J) F, G) D)

{Preorder, sometimes called dynastic order for selecting the successor to the throne in England, etceteras}

**Traverse the natural correspondence in preorder:**

ABCKDEHFJG -- Note the same as a preorder traversal of the forest.

**Traverse the natural correspondence in inorder:**

BKCAHEJFGD -- Note the same as postorder for the forest.

# Artificial Intelligence

## Game of 16

|    |    |    |    |
|----|----|----|----|
| 1  | 2  | 3  | 4  |
| 5  | 6  | 7  | 8  |
| 9  | 10 | 11 | 12 |
| 13 | 14 | 15 | =  |

As a child, many people learn to play the game of 16. The above pattern is frequently considered the winning combination.

The game consists of 15 tiles and one space. An “=” is used in the above diagram to represent the space. On each move, the player exchanges the location of the hole (space) with an adjacent tile.

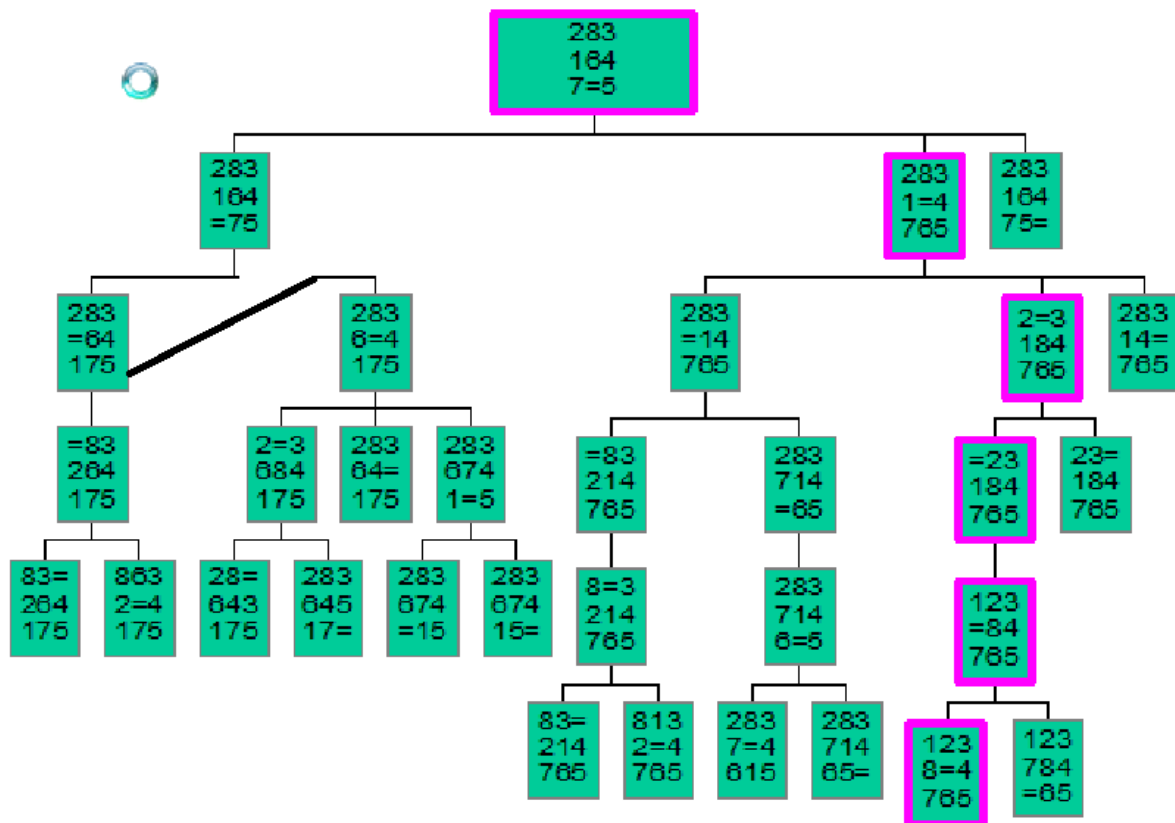
The game of 16 can be used as a simple example of game theory in Artificial Intelligence (AI). First the tiles are moved to random positions. The player must then move the tiles to a predetermined pattern to complete (win) the game. While the previous pattern is the most popular winning combination, other patterns are also popular.

The game of 16 is difficult to solve by hand. We will consider the game of 8 with eight tiles and a space as a simplification. Again, an “=” will be used to represent the space.

- Assume the starting position for the tiles on the top left diagram.
- The game is won when the tiles are arranged as in the diagram on the bottom left.

|   |   |   |
|---|---|---|
| 2 | 8 | 3 |
| 1 | 6 | 4 |
| 7 | = | 5 |

|   |   |   |
|---|---|---|
| 1 | 2 | 3 |
| 8 | = | 4 |
| 7 | 6 | 5 |



The “game tree” is constructed one level at a time by moving the “space” one tile in each direction it has not yet been from its current location. This constitutes all possible new positions that can be reached in one turn. Note there are three possible moves from the starting position, i.e., one position to the left, one position up, and one position to the right.

The game may be won by exhaustively attempting all combinations till the winning board is encountered. The correct set of winning moves may be determined by tracing the path from the top (root) of the tree to the winning game board. This path has been high-lighted in the diagram.

Compare this to playing chess, checkers, and other board games. At each turn, a player could build a similar “game tree” for each of their pieces on the board. Each level of the tree would represent what could be accomplished in one turn. Note that this same strategy could be used to drive a computer driven war simulation, or Star Wars Defense System (anti-ballistic missile defense system).

In chess, checkers, and similar games, the person with first move always has the advantage. They must make a mistake for the second player to have a chance. If the player with first move can build the entire game tree (their move and their opponents) prior to each move, they should always win. **Then why don't computer chess and checker games always win?**

The answer is in how fast the game tree grows. Note the exponential growth of the tree in the game of 8. On each move there is a maximum of only three adjacent squares in which to move. In many board games there are 20 to 30 pieces, and each piece may move 20 or 30 different places. The number of possibilities at each turn rapidly grows to hundreds of millions which over whelms most computers. The normal solution is to build as many levels of the game tree in the allotted time (or allotted resources) as possible. The best move to “that point in time” is then made. **In chess each player has approximately 30 moves for a combined total of 900 positions. A standard game has about  $10^{125}$  board positions. Evaluated at 1 billion movers per second it would require about  $10^{108}$  years to evaluate all positions.** The individual with first move should always win, i.e, the second player can only win or draw if the first player makes a mistake. An algorithm assuring the win for the first player has been know for over 100 years. Computer chess games play “book” at the start and end game.

The **artificial intelligence** is in how the computer determines what constitutes the best move encountered. The optimum move is winning this turn. If not this turn, then the next turn while not losing to the opponent on their turn. If you can not win, then pick the move that causes the opponent the most damage and / or provides the best future board position.

For example in checkers, a king is worth more than a single checker. All game pieces in chess have an associated value (pion-1, knight-2, bishop & rook/castel-3, queen-5, king-beyond value). In both checkers and chess, control of the center squares on the game board is more important than squares on the side. More game pieces can attack through the center than down a side. Formulating a strategy the computer can follow constitutes the artificial intelligence. This is frequently a mathematical formula that is a function of each game pieces individual values, their current location, and perceived strategic value.

Again consider the game of 16. Lets assume that our computer can only generate 2 levels in the tree each turn and that our opponent can generate 4 levels. We will probably lose most of the time. Our opponent looks farther ahead each time they make a choice.

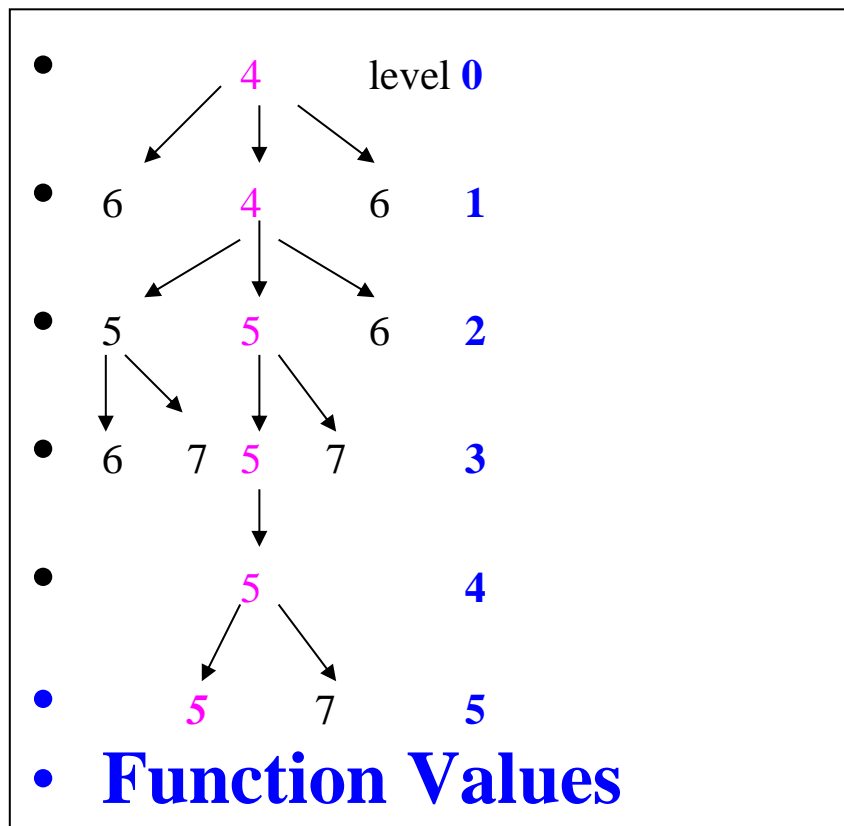
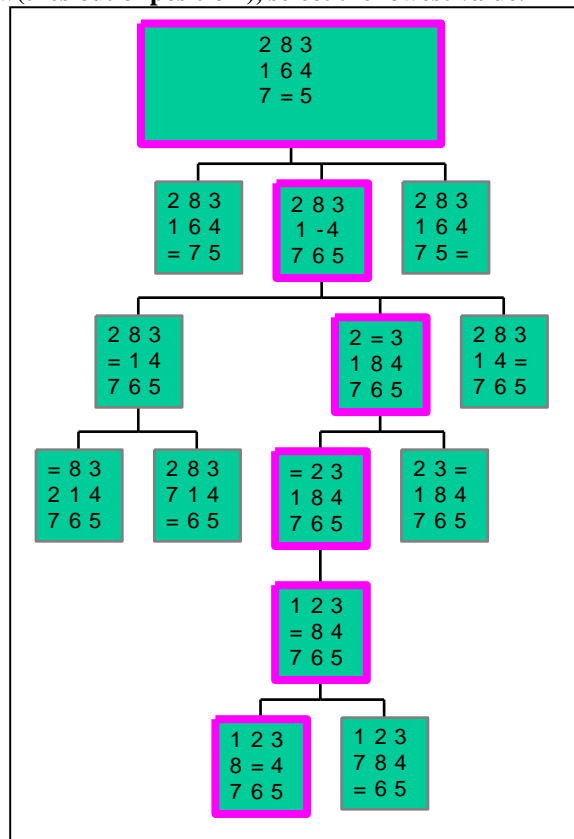
Note that most branches of the tree are dead ends. If we could eliminate those dead ends at an earlier point, the time not wasted in generating them could be used to drive more promising branches deeper. Perhaps of 6 or 7 levels deep. This would improve our chances of winning.

What should we consider in arriving at this formula (AI). The depth in the game tree to arrive at a desirable goal would be important. Note each level of the tree represents an increase of depth (moves to reach) of one. In general the fewer moves required, probably the better. In addition, the greater the number of tiles out of order from the winning combination, probably the worse the move.

Consider a function to be used at each move (level) as follows:

**Best Move [BM] = f(level) + w(tiles)**, where f(level) is the number of levels (moves) into the tree and w(tiles) is a count of tiles out of order. At each level we will chose to discard all but the lowest value for BM. If there is a tie at any level for the lowest value of BM, then each node in the tie must be expanded another level. When a best move (lowest value for BM) is discovered, discard all intermediate nodes in the tree that were generated. Note that for each expansion of the following tree (left) the corresponding values of BM have been computed and displayed on the right. For example, the value of BM for the root node (starting point) is  $f(\text{levels}) + g(\text{tiles}) = 0 + 4 = 4$ . The values for the next level are from left to right  $BM = 1+5=6$ ,  $1+3=4$ , and  $1+5=6$ . The two branches with values of 6 are discarded and only the branch of four is expanded. At level 2 there is a tie (two 5's) which must both be expanded. At level three it is possible to select a winner. The losing branches are discarded.

Best Move [BM] =  $f(\text{level}) + w(\text{tiles out of position})$ , select the lowest value.

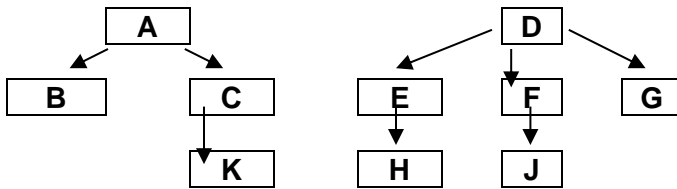


At level 5 the winning board position is encountered. The correct moves to complete the game may be determined by tracing back up the tree to our starting position (the root). If the winning combination had not been encountered at level 5, we would have picked as our next move to level 1 the move that produced the most advantageous game position in the time allotted for the move. Note that in this example, f(levels) is a red herring. It does not help in determining the winning path (tree branch). Checkers and chess are similar, there are just more game pieces with which to build the tree.

Note that this same idea could be used to wage war on a modern battle field. The loss of planes, soldiers, and strategic position would be used to compute the next “best move.” The code for the Strategic Defense Initiative (Star Wars space platform to shoot down ICBM’s) might use this type strategy. The software would have to apply its AI to determine what is really an attacking missile or plane as opposed to normal air traffic, and meteorites hitting the atmosphere.

(Robert Hyatt USM 1975 through 1976 – 9<sup>th</sup> ranked chess program in world on 1.5 MIPS Xerox Sigma 9 against computers like Cray Research/Control Data Corporation at 66 MIPS).

-- in file AITem.doc



|   |   |   |   |   |   |   |   |   |    |
|---|---|---|---|---|---|---|---|---|----|
| 0 | 0 | 1 | 2 | 0 | 1 | 0 | 1 | 0 | 3  |
| B | K | C | A | H | E | J | F | G | D  |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

### Postorder by Degrees:

Assume a game like chess or checkers. We would like to implement a program to play the game (artificial intelligence). Let the moves that can be made by each piece be represented by a tree, the collection of trees would form the forest of possible moves. The number of trees in the forest is the number of pieces on the game board. The possible moves a piece X can make is represented by his sons, their moves by their sons, and so on. At each move a locally defined function is applied to select the next best move from the available alternatives. The function (heuristic) normally considers such things as the value of board positions controlled, value of game pieces taken or threatened, or any other factor resulting in an advantage or disadvantage.

### Algorithm NextMove:

Assume  $f$  is a function of the nodes of a tree such that  $f$  at a node  $x$  depends only on  $x$  and the values of  $f$  on the sons of  $x$ . The following algorithm evaluates  $f$  at each node (game piece) of a nonempty forest and returns the "value" of all moves possible for each piece in a stack. The number of items in the stack at termination is the number of available game pieces {one stack entry for each tree in the forest}. The forest must be represented in postorder by degrees.

- 0) Set the stack empty and let  $P$  point to the first node of the forest in postorder.

Loop

- 1) Set  $d$  to the Degree of  $P$  ( $d \leftarrow P.\text{Degree}$ ). Evaluate  $f(P.\text{Node})$ , using the values  $f(x_d), \dots, f(x_1)$  found on the stack. {The value of node  $P$  is a function of himself and the value of his sons.}
- 2) Remove the top  $d$  items from the stack and replace them with the value of  $f(P.\text{Node})$  {Push the value of the function applied to node  $P$  into the stack replacing the values of  $P$ 's sons.}.
- 3) If possible
  - set  $P$  to  $P\$$  {the next node in postorder,  $P \leftarrow P + 1$  in the above representation}
  - else
  - Exit Loop
  - end if

End Loop;



## Algorithm Equivalence:

Let  $S$  be a set of items numbered 1 through  $N$ ,  $\{1, 2, \dots, N-1, N\}$ , and let  $\text{Parent}(1), \dots, \text{Parent}(N)$  be a table of integers. This algorithm accepts a set of equivalence relations of the form " $j = k$ " (read  $j$  is equivalent to  $k$ ) and adjusts the  $\text{Parent}$  table to represent a set of trees, so that two elements are equivalent if and only if they belong to the same tree.

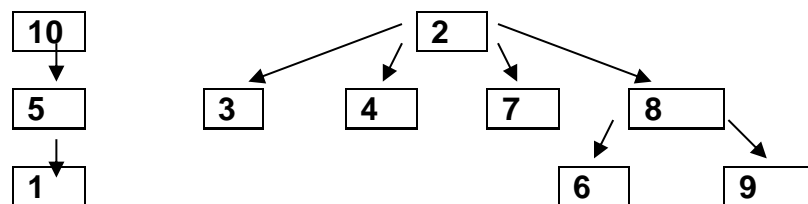
- 1) For  $I$  in  $1..N$  loop  $\text{Parent}(I) := 0$ ; end loop;
- 2 Get the first relation " $J = k$ " from the input stream.  
While (not end-of-file on the input stream) loop  
    {Find each item's most distant parent}  
    While ( $\text{Parent}(J) > 0$ ) loop  $J := \text{Parent}(J)$ ; end loop;  
    While ( $\text{Parent}(K) > 0$ ) loop  $K := \text{Parent}(K)$ ; end loop;  
    If  $J \neq K$  then  $\text{Parent}(J) := K$ ; end if;  
    Get next relation " $J = K$ " or set end-of-file to true.  
end loop;

Example, assume the following equivalence relations for 10 items:

$1 = 5, 6 = 8, 7 = 2, 9 = 8, 3 = 7, 4 = 2, 9 = 3$ , and  $1 = 10$ .

After processing the relations the resulting table and trees:

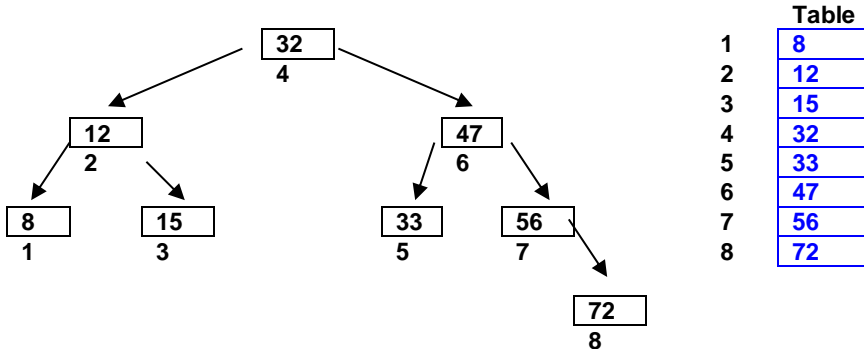
|           |   |   |   |   |    |   |   |   |   |    |
|-----------|---|---|---|---|----|---|---|---|---|----|
| Parent(K) | 5 | 0 | 2 | 2 | 10 | 8 | 2 | 2 | 8 | 0  |
| K         | 1 | 2 | 3 | 4 | 5  | 6 | 7 | 8 | 9 | 10 |



The relations might represent jobs at a construction site where management wants to assign teams equivalent work loads to different construction teams. Algorithm Equivalence is a powerful tool when combined with topological sorting to determine critical paths.



## Binary Search



### Algorithm B1 (Binary Search):

Assume an array of  $N$  records  $R_1, R_2, \dots, R_N$  whose keys are in increasing order  $K_1 < K_2 < \dots < K_N$ . This algorithm searches for a record with key  $Key$ . Let  $LB$  mean lower bound of the current table and  $UB$  mean the upper bound of the current table.

Set  $LB := 1$ ;  $UB := N$ ;

loop

    exit when  $UB < LB$  --Terminate, the search is unsuccessful.

$I := \lfloor (LB + UB) / 2 \rfloor$ ;

    if  $Key < K_I$  then

$UB := I - 1$ ;

    elseif  $Key > K_I$  then

$LB := I + 1$ ;

    else -- note:  $Key = K_I$  then

        Process the record;

        Exit;

    end if;

end loop;

Note that each time you move right or left in a balanced tree you eliminate  $\frac{1}{2}$  the remaining possibilities. Hence the maximum number of levels for a tree containing  $N$  keys is approximately  $\log_2 N$ . Assume all keys have an equal probability of being the next desired key on a search. The minimum number of probes to find a random key is 1. The maximum number of probes to locate a key is the number of levels in the tree, approximately  $\log_2 N$ . The average must be between the minimum and maximum. Looking at a few trees we quickly discover most of the keys are located either on the bottom level or preceding level. This implies the average (expected) number of probes is almost identical to the maximum, i.e.,  $O(\log_2 N)$ .

**Note a sequential search of a table consisting of about 64 items may be faster than a binary search. The sequential search will make approximately  $(N+1)/2$**

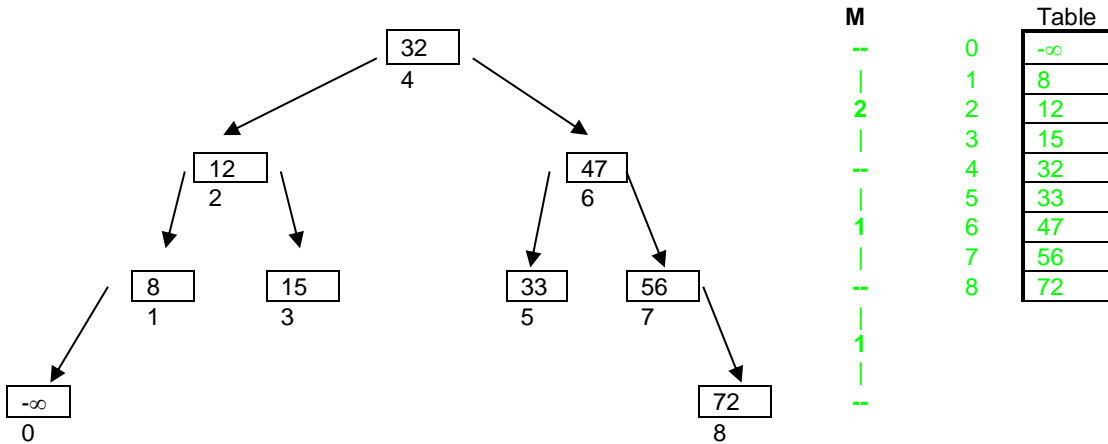
**probes or 32.5 probes. A binary search would only make  $\log_2 N$  or 6 probes.**

**But the cost per probe is higher.** In a sequential search, we need only add a constant offset to reach the next search location. For the binary tree we must first decide which half to throw away. Then we must move boundaries and calculate a new mid point. The cost per probe for the binary search is substantially higher than for the sequential search. Put another way, as the expected number of probes in a sequential search increases, it will eventually exceed the overhead experienced by the binary search which makes far fewer probes in a large table. As a rule of thumb, a sequential search is faster for tables with fewer than 50 items and a binary search faster for larger tables.

**To make the binary search run faster we must reduce the overhead.** Consider 8 / 2. The divide instruction on most computers is expensive. Even with a hardware assist it is up to 30 times slower than shift and logical instructions (“and” and “or”) on most computers. Note if we shift the binary 8 in base 10 right by one bit in base 2, the result is 4, e.g.  $8_{10} = 1000_2$  shifted right one bit is  $0100_2 = 4_{10}$ . The expression  $A/256$  can be accomplished by a right shift of 8 bits as  $256 = 2^8$ . Hence dividing (right shift) and multiply (left shift) by numbers that are a power of two using shift operations is up to 30 times faster than traditional divide and multiply operations. Using a shift instruction to replace the divide operation in the binary search will make it run substantially faster.

Over head can be further reduce eliminating tracking of the exact position of the base and bounds as shown below in algorithm B2.

Binary searches are most effective when the contents of the table is static. Consider the overhead associated with frequent random insertions and deletions and the restriction of keeping the table in sorted order.



### Algorithm B2 (Uniform Binary Search):

Assume an array of  $N$  records  $R_1, R_2, \dots, R_n$  whose keys are in increasing order  $K_1 < K_2 < \dots < K_n$ . This algorithm searches for a record with key  $Key$ . If the number of items in the table  $N$  is even, the algorithm will sometimes refer to a dummy key  $K_0$  which should be set to a value less than any value of  $K$  that could appear in the table. Record 0 exists strictly to facilitate the search, e.g. try a tree with  $N = 10$ .

Set  $I := \lceil N/2 \rceil$ ;  $M := \lfloor N/2 \rfloor$ ;

loop

    if  $Key < K_I$  then

        if  $M = 0$  then

            exit; -- The search terminates unsuccessfully.

        else

$I := I - \lceil M/2 \rceil$ ;

$M := \lfloor M/2 \rfloor$ ;

        end if;

    elsif  $Key > K_I$  then

        if  $M = 0$  then

            exit; -- The search terminates unsuccessfully.

        else

$I := I + \lceil M/2 \rceil$ ;

$M := \lfloor M/2 \rfloor$ ;

        end if;

    else -- Note  $Key = K_I$

        Process the record;

        Exit; -- The search was successful.

    end if;

end loop;

Alternate: store offsets in an array – requires a memory access (main versus cache memory): divide by 2 using right shift one bit.

### III Spelled Name Routines:

CACM 5(1962) pp. 169-171, by Davidson.

|  |  |  |  |  |
|--|--|--|--|--|
|  |  |  |  |  |
|--|--|--|--|--|

Five character code:

- 1) 1<sup>st</sup> character is 1<sup>st</sup> character of surname name.
- 2) 5<sup>th</sup> character is 1<sup>st</sup> character given name, if not known use the wild card "\*"
- 3) The 2<sup>nd</sup>, 3<sup>rd</sup> and 4<sup>th</sup> character are initialized to spaces.
- 4) Using the surname:
  - A) Strike out all vowels, H, W, and Y.
  - B) Squeeze letters.
  - C) Strike out all but one occurrence of repeated letters.

Example:

David Miller:

Eliminate vowels, H, W, and Y:

M    ĩ    l    l    e    r

Eliminate repeated letters:

M    l    ĩ    r

Yields:

|   |   |   |  |   |
|---|---|---|--|---|
| M | l | r |  | D |
|---|---|---|--|---|

Note that Miller and Muller produce the same storage key.

|             |    |               |
|-------------|----|---------------|
| McGone, W   | -> | McKone, W     |
| Buckmann, R | -> | Bucknam, R    |
| Curbie      | -> | Kirby         |
| Iggins, R   | -> | Higgins, R    |
| Angreiff    | -> | Ungry, Singer |

Example: Buris, Burris, Burrows, Buris, Burres; Hebert or Abear

**“Soundex”** method by Margaret K. Odell and Robert C. Russell [cf. U.S. Patents 1261167 (1918), 1435663 (1922)] for surnames:

- 1) Retain the first letter of the name and drop all occurrences of a, e, h, i, o, u, w, and y in all other positions.
- 2) Assign the following numbers to the remaining letters after the first:  
b, f, p, v => 1  
c, g, j, k, q, s, x, z => 2  
d, t => 3  
l => 4  
m, n => 5  
r => 6
- 3) If two or more letters with the same code were adjacent in the original name (before step 1), omit all but the first.
- 4) Convert to the form “letter, digit, digit, digit” by adding trailing zeros (if there are less than three digits), or by dropping the rightmost digits (if there are more than three).

Example:

Roger        Rgr    R26

Rogers      Rgrs   R262

Rogger      Rggr   R26

Euler            E460

Gauss            G200

Hilbert          H416

Lukasiewicz      L222

See C. P. Bourne and D. F. Ford, JACM 8(1961), pp. 538-552; Leon Davidson, CACM 5 (1962), pp. 169-171; Federal Population Census 1790-1890 (Washington, D.C.: National Archives, 1971), 90.

Treat all keys in a database producing equivalent soundex code as a single list for retrieval. Use additional information in the records to select the unique or group of keys desired.

Example: Burris, Buris, Burrows, Buris, Bures

-- in file ILLSPELL.DOC

## Lists Ordered by Frequency of Use

Assume a list of  $N$  items is maintained in random order and searched sequentially. The minimum number of probes to find a random item in the list is 1 and the maximum number of probes is  $N$ . If all items in the list have an equal probability of reference, then the expected number of probes to find a random item is:

$$E_{\text{probes}} = 1p_1 + 2p_2 + 3p_3 + \dots + Np_N$$

where  $p_i$  is the probability that the  $i$ th object is the item desired. We have assumed equal probabilities, hence  $p_i = 1 / N$  for all  $i$ . This implies

$$E_{\text{probes}} = (1 + 2 + 3 + \dots + N) * (1/N) = (N*(N+1)/2) * (1/N) = (N + 1) / 2.$$

W. P. Hensing reported in IBM Systems Journal {2(1963), p114-115} that many commercial applications obey an “80-20” rule. Eighty percent of the transactions are restricted to twenty percent of the stock, and the same rule may be applied recursively to transactions on the most active twenty percent of stock. In other words 64% of the transactions are restricted to the most active 4% of stock. (A Pareto distribution is also used to explain this phenomenon.) If a list is ordered in descending order based on frequency of use and searched sequentially, then the expected number of probes reduces to approximately  $E_{\text{probes}} = 0.122N$ , which represents about a factor of four improvement over the  $(N+1)/2$  probes required for random permutations.

List ordered by frequency of use may exhibit better search performance than the binary search and other search techniques. Note that while the list order by frequency of reference may require more average probes to locate an item than a binary search it may still be faster. If the current item is not the desired item in a sequential search, you only need to look in the next position. In a binary search you must decide which half of the list to eliminate then calculate a new mid point in the remaining list. Hence the overhead per probe in a binary search is higher than the overhead per probe in a sequential search.

Example: Assume 8 items with frequencies of reference = 80%, 14%, 1%, 1%, 1%, 1%, 1%, and 1%. Then  $E_{\text{probes}} = 1(0.8) + 2(0.14) + 3(0.01) + 4(0.01) + 5(0.01) + 6(0.01) + 7(0.01) + 8(0.01) = 0.8 + 0.28 + 0.03 + 0.04 + 0.05 + 0.06 + 0.07 + 0.08 = 1.41$ . Hence the expected number of probes for a random item in the list is 1.41. Eighty percent of request are satisfied in one probe and 94% of requests in two probes. A binary search would require 3 probes and sequential search 4.5 probes.

### Self Organizing Lists

Lists that adjust the order of their entries dynamically based on use.

### AVL or Height Balanced Trees



## B-Tree Definition

A B-tree of order  $m$  is a tree satisfying the following properties:

- 1) Every node has  $\leq m$  sons.
- 2) Every node, except for the root and the leaves, has  $\geq m/2$  sons.
- 3) The root has at least 2 sons, else it is a leaf.
- 4) All leaves appear on the same level and carry no information.
- 5) A non-leaf node with  $k$  sons contains  $k-1$  keys.

B-trees were discovered by R. Bayer and E. McCreight in 1972, Acta Informatica, pp. 173-189. M. Kaufman discovered them independently but did not publish his work.

Format for a node of order 5 with  $J = 4$  keys and  $J + 1 = 5$  pointers:

|     |       |     |       |     |       |     |       |     |
|-----|-------|-----|-------|-----|-------|-----|-------|-----|
| Pt0 | Key 1 | Pt1 | Key 2 | Pt2 | Key 3 | Pt3 | Key 4 | Pt4 |
|-----|-------|-----|-------|-----|-------|-----|-------|-----|

B-tree of order 5. Note on split,  $3 \leq \text{number of children} \leq 5$  and  $2 \leq \text{number of keys} \leq 4$  in each node unless it is the root.

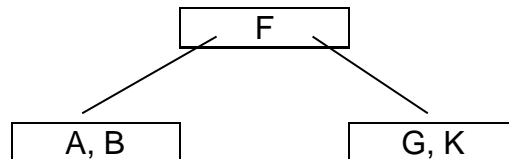
Insert A, B, G (in any order):

|         |
|---------|
| A, B, G |
|---------|

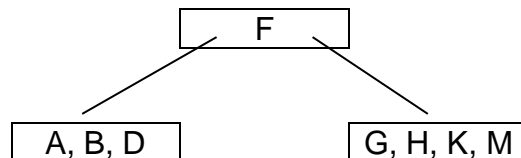
Insert F (note contents maintained in sorted order):

|            |
|------------|
| A, B, F, G |
|------------|

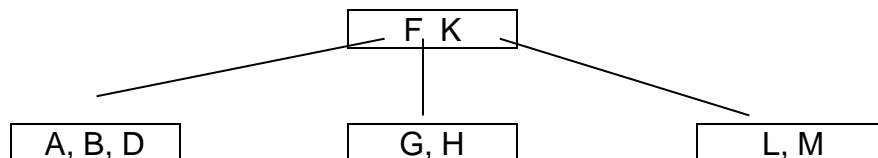
Insert K:



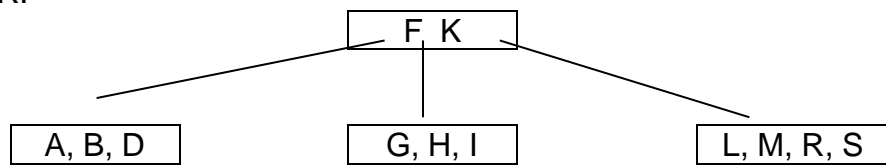
Insert D, H, M:



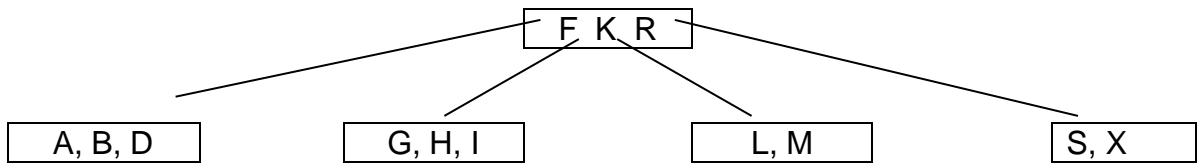
Insert L:



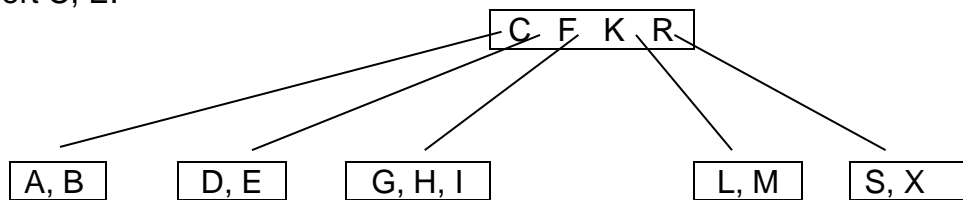
Insert S, I, R:



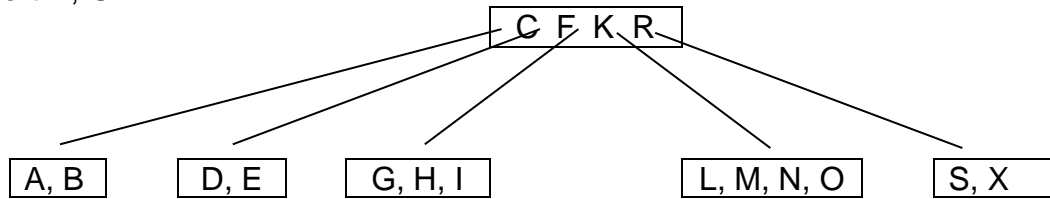
Insert X:



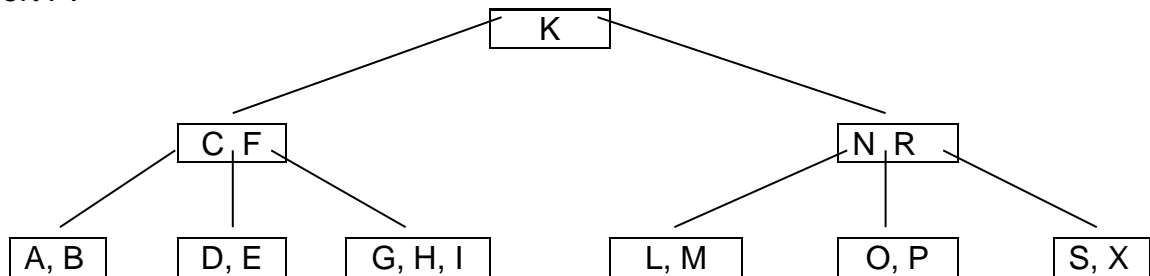
Insert C, E:



Insert N, O:



Insert P:



Assume we wish to access a very large disk file both randomly and sequentially in ascending storage key order. We desire that the storage key be alphanumeric. Assume a B-tree is used to store the record keys along with a pointer to the storage address. Note that we measure CPU time in pico or nano seconds. Disk access time to fetch portions of a very large directory or a record is measured in milliseconds. We clearly wish to limit the number of disk accesses. Pick the B-tree node size to correspond to a "page" of disk space. Assume that a page is the unit of space read and written by the disk access mechanism. Now consider worst case access when searching a B-tree of order  $m$  where  $m$  corresponds to the maximum size B-tree node that will fit in one disk page. Assume this allows for  $N$  keys. Then there are  $N+1$  null pointers in the leaves of the tree on level  $l$ . Assume the root is at level 0 in the tree. Then the number of nodes on levels 1, 2, 3, is at least 2,  $2\lceil m/2 \rceil$ ,  $2\lceil m/2 \rceil^2$ , etceteras. Remember that each node except the root must be at least 50% full. This implies that  $N+1 \geq 2\lceil m/2 \rceil^{l-1}$ .

**Hence  $l \leq 1 + \log_{\lceil m/2 \rceil} ((N+1)/2)$ . As an example, assume  $N = 1,999,998$  and  $m = 199$ . Then  $l$  is at most 3 disk accesses (tree levels) to find any random key.** The following program uses a binary search within a memory page (node) to locate a key or pointer to the next level, hence the maximum search time within a memory node is approximately proportional to  $\log_2 199$  or 14.2.

Many refinements have been suggested to improve performance. For example, [when a node is already full and must except a new key, see if the key can be placed in a sibling node on the left or right before splitting the node](#). This guarantees 2/3 storage utilization as the minimum. This is normally called a B\* tree. Another suggestion is [use the space occupied by pointers in leaves to store additional information](#). Note all pointers in leaves are null and represent the majority of pointers.

### Sample deletions:

**Start:**

f j  
a d  
g h  
k l s

**delete s**

f j  
a d  
g h  
k l

**delete l**

f  
a d  
g h j k

**delete a**

g  
d f  
h j k

**delete f**

h  
d g  
j k

**delete h**

d g j k

**Start:**

f j  
a d  
g h  
k l s

**delete f**

j  
a d g h  
k l s

**delete j**

h  
a d g  
k l s

**delete l**

h

a d g  
k s

**delete s**

g  
a d  
h k

**delete k**

a d g h

```

-- In file Btree.ads
-- This package creates a B-Tree of order TreeOrder. TreeOrder should
-- be an even integer. TreeOrder represents a tree with TreeOrder - 1
-- keys and TreeOrder pointers. The elements of the tree are of type
-- MyKey. MyKey may be any intrinsic or programmer defined type. If
-- MyKey is a programmer defined type, then overloads must be supplied
-- for the functions ">=" and "<=." In addition, an overload for
-- procedure "put" must be supplied that will print elements of
-- type MyKey (or at least the desired portion of MyKey).

```

**generic**

```

    TreeOrder: Positive; --Must be an even integer to work correctly.
    type MyKey is private;
    with function "<=" (P1,P2: MyKey) return Boolean;
    with function ">=" (P1,P2: MyKey) return Boolean;
    with procedure put (P1: MyKey);

```

**package BTree is**

```

    -- Search for SearchKey in B-tree created by this package.
    -- If found display an appropriate message. If not found,
    -- insert the key in the tree.

```

```

    procedure InsertBTree (SearchKey: in MyKey);

```

```

    -- Deletes the SearchKey from the tree if it exists.

```

```

    procedure DeleteBtreeKey (SearchKey: in MyKey);

```

```

    -- Prints the contents of the tree in ascending key order.

```

```

    procedure PrintBtree;

```

**end BTree;**

```

-- In file Btreeu.adb
with Ada.Text_IO; use Ada.Text_IO;
with Btree;
procedure BTreeU is

    Ch: Character;
    package BTreeChar is new BTree(4, Character, "<=", ">=", put);
    use BTreeChar;

    type MonthType is
        (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);

    type DateType is record
        Month: MonthType;
        Day: Integer range 1..31;
        Year: Integer range 1700..2400;
    end record;

    Date: DateType;

    package MonthTypeIO is new
        Ada.Text_IO Enumeration_IO (MonthType); use MonthTypeIO;
    package IntegerIO is new
        Ada.Text_IO Integer_IO (Integer); use IntegerIO;

    function "<=" (P1, P2: DateType) return Boolean is
    begin
        if P1 = P2 then return true; end if;
        if P1.Year < P2.Year then return true; end if;
        if P1.Year = P2.Year and then P1.Month < P2.Month then
            return true; end if;
        if P1.Year = P2.Year and then P1.Month = P2.Month
            and then P1.Day <= P2.Day then return true;
        else
            return false;
        end if;
    end;

    function ">=" (P1, P2: DateType) return Boolean is
    begin
        if P1 = P2 then return true; end if;
        if P1.Year > P2.Year then return true; end if;
        if P1.Year = P2.Year and then P1.Month > P2.Month then
            return true; end if;
        if P1.Year = P2.Year and then P1.Month = P2.Month
            and then P1.Day >= P2.Day then return true;
        else
            return false;
        end if;
    end;

    procedure put(aDate: DateType) is

```

```

begin
    put(aDate.Month); put(" "); put(aDate.Day,3); put(aDate.Year,5);
end put;

package BTreeDate is new Btree(4,DateType,"<=",">=",put);
use BTreeDate;

begin

    put("Enter a character, 'Z' to stop "); get(Ch);
    while ch /= 'Z' loop
        BTreeChar.InsertBTree(Ch);
        BTreeChar.PrintBTree; new_line;
        put("Enter a character, 'Z' to stop "); get(Ch);
    end loop;
    new_line(3);

    loop
        put("Enter char to delete, Z to stop: "); get(Ch);
        exit when Ch = 'Z';
        BTreeChar.DeleteBtreeKey(Ch);
        BTreeChar.PrintBTree;
    end loop;

    Date :=(Jan, 15, 1947);
    BTreeDate.InsertBTree(Date);
    BTreeDate.InsertBTree((Feb, 14, 1892));
    BTreeDate.InsertBTree((Jan, 16, 1947));
    BTreeDate.InsertBTree((Sep, 14, 1892));
    BTreeDate.InsertBTree((Jan, 15, 2020));
    put("Date tree is built"); new_line;
    BTreeDate.PrintBTree; new_line;
    put("Delete Feb 14, 1892:"); new_line;
    BTreeDate.DeleteBTreeKey((Feb, 14, 1892));
    BTreeDate.PrintBTree;
end BTreeU;

```

```

-- Builds a BTree of order M (M = TreeOrder, must be an even number).
with Ada.Text_IO; use Ada.Text_IO;
package body BTree is
  M: constant integer := TreeOrder;
  Mdiv2: constant integer := M/2;

  type Page;
  type PagePt is access Page;
  type Item is record
    Key: MyKey;
    Pt: PagePt;
    DiskPt: integer; --Normally a pointer to disk for record.
  end record;

  type KeyArray is array(1..M) of Item;
  type Page is record
    NumItems: integer range 0..M;
    P0: PagePt;
    KeyEntry: KeyArray;
  end record;

  Root: PagePt := null; -- Pointer (root) to tree built by package.

  -- Search for SearchKey in B-tree with root aTree. If found display
  -- an appropriate message. If not found, insert the key in the
  -- tree. If a node splits, indicate this to the parent by setting
  -- "Split" to true and passing a pointer "MiddleKey" back
  -- to the parent.
  --
  procedure Search(SearchKey: in MyKey; aTree: in out PagePt;
    Split: in out Boolean; AnEntry: in out Item) is

    K, L, R: Integer;
    Q: PagePt;
    MiddleKey: Item;

    procedure Insert is -- Insert key in this page.
      SplitNode: PagePt; --R and MiddleKey are defined globally in Search.
    begin
      -- Insert middle key to right of aTree.KeyEntry(r).
      if aTree.NumItems < M then --Is there space in this node for the new key?
        aTree.NumItems := aTree.NumItems + 1;
        Split := false;
        for I in reverse (R+2)..aTree.NumItems loop -- Move keys up for insert.
          aTree.KeyEntry(I) := aTree.KeyEntry(I-1);
        end loop;
        aTree.KeyEntry(R+1) := MiddleKey;
      else -- must insert key in another page, select the middle key.
        SplitNode := new Page;
        if R <= Mdiv2 then
          if R = Mdiv2 then -- New key is middle key after the split.
            AnEntry := MiddleKey;
          else
            AnEntry := aTree.KeyEntry(Mdiv2); -- Insert new key in left page.
            for I in reverse (R+2)..Mdiv2 loop
              aTree.KeyEntry(I) := aTree.KeyEntry(I-1);
            end loop;
          end if;
        else
          AnEntry := SplitNode.KeyEntry(1);
          SplitNode.P0 := aTree.P0;
          SplitNode.KeyEntry(2) := MiddleKey;
          SplitNode.NumItems := 2;
          aTree.P0 := SplitNode;
          Split := true;
        end if;
      end if;
    end Insert;
  end Search;
end BTree;

```

```

        end loop;
        aTree.KeyEntry(R+1) := MiddleKey;
    end if;
    for I in 1 .. Mdiv2 loop
        SplitNode.KeyEntry(I) := aTree.KeyEntry(I+Mdiv2);
    end loop;
else -- insert new key in right page
    R := R - Mdiv2; AnEntry := aTree.KeyEntry(Mdiv2+1);
    for I in 1..(R-1) loop
        SplitNode.KeyEntry(I) := aTree.KeyEntry(I+Mdiv2+1);
    end loop;
    SplitNode.KeyEntry(R) := MiddleKey;
    for I in (R+1)..Mdiv2 loop
        SplitNode.KeyEntry(I) := aTree.KeyEntry(I+Mdiv2);
    end loop;
end if;
aTree.NumItems := Mdiv2; SplitNode.NumItems := Mdiv2;
SplitNode.P0 := AnEntry.Pt;
AnEntry.Pt := SplitNode;
end if;
end Insert;

begin -- Start Search procedure.
if aTree = null then -- Return signalling to add a new key.
    Split := true; --Force call to insert and potential page split.
    AnEntry.Key := SearchKey;
    -- set disk pointer in AnEntry.PagePt to disk address.
    AnEntry.Pt := Null;
else -- Look for path to key using a binary search of page (node).
    L := 1; R := aTree.NumItems;
    loop
        K := (L+R)/2; -- Integer divide.
        if SearchKey <= aTree.KeyEntry(K).Key then R := K - 1; end if;
        if SearchKey >= aTree.KeyEntry(K).Key then L := K + 1; end if;
        exit when R < L;
    end loop;

    if L - R > 1 then
        put("The key already exists in the tree."); new_line;
        Split := false;
    else
        if R = 0 then
            Q := aTree.P0; -- Search to left of node.
        else
            Q := aTree.KeyEntry(R).Pt; -- Search to right of key at location R.
        end if;
        Search(SearchKey, Q, Split, MiddleKey);
        if Split then Insert; end if; -- Key not found, must insert.
    end if;
end if;
end Search;

-- Prints contents of the tree.
procedure PrintTree(P: PagePt; L: Integer) is
begin
    if P /= null then

```



```

    for I in 1 .. L loop put(" "); end loop;
    for I in 1 .. P.NumItems loop put(" "); put(P.KeyEntry(I).Key); end loop;
    new_line;
    PrintTree(P.P0, L+1);
    for I in 1 .. P.NumItems loop PrintTree(P.KeyEntry(I).Pt, L+1); end loop;
  end if;
end PrintTree;

-- Invokes procedure to print the contents of the tree in ascending
-- order starting at the root.
--
procedure PrintBTree is
begin
  PrintTree(Root,1);
end;

-- Primary procedure to build the B-tree. It accepts the SearchKey
-- and initiates the search. If the root node splits, then this
-- procedure creates a new root node and inserts the key. If the
-- tree is empty at the start (no keys), then it is treated as an
-- existing root node that split. The pointers to the left and right
-- of the key inserted in the root are just null. Note that Ada
-- automatically initializes pointer variables to null. The assignment
-- of null to new pointers in the program is to emphasize they must
-- be set to this value (if the program is translated to another
-- language). We assume the basic unit for disk allocation is a page.
--
procedure InsertBTree(SearchKey: in MyKey) is
  TempPage: PagePt;
  Split: Boolean;
  AnEntry: Item;
begin
  Search(SearchKey, Root, Split, AnEntry);
  if Split then -- A new root page has been created, initialize it.
    TempPage := Root;
    Root := new Page;
    Root.NumItems := 1;
    Root.P0 := TempPage;
    Root.KeyEntry(1) := AnEntry;
  end if;
end InsertBTree;

procedure delete(DeleteKey: in MyKey; TreePt: in out PagePt; TooSmall: in out Boolean) is
  k,l,r: integer;
  TempPage: PagePt;

  procedure underflow(c,TreePt: in out PagePt; s: in integer; TooSmall: in out boolean) is
    -- TreePt is underflow page and c is the parent page. Pages must be combined.
    b: PagePt;
    k,mb,mc, TempS: integer;
  begin
    TempS := S;
    mc := c.NumItems; -- TooSmall = true, TreePt.NumItems = Mdiv2 - 1.
    if TempS < mc then -- b is the page to the right of TreePt.
      TempS := TempS + 1;
      b := c.KeyEntry(TempS).Pt;
    end if;
  end underflow;
end delete;

```

```

mb := b.NumItems;
k := (mb-Mdiv2+1) / 2;
-- k is the number of items available on the adjacent page b.
TreePt.KeyEntry(Mdiv2) := c.KeyEntry(TempS); TreePt.KeyEntry(Mdiv2).Pt := b.P0;
if k > 0 then -- Move k items from page b to page TreePt.
  for l in 1 .. (k-1) loop TreePt.KeyEntry(l+Mdiv2) := b.KeyEntry(l); end loop;
  c.KeyEntry(TempS) := b.KeyEntry(k); c.KeyEntry(TempS).Pt := b;
  b.P0 := b.KeyEntry(k).Pt; mb := mb - k;
  for l in 1 .. mb loop b.KeyEntry(l) := b.KeyEntry(l+k); end loop;
  b.NumItems := mb; TreePt.NumItems := Mdiv2 - 1 + k; TooSmall := false;
else -- merge pages TreePt & b.
  for l in 1 .. Mdiv2 loop
    TreePt.KeyEntry(l+Mdiv2) := b.KeyEntry(l);
  end loop;
  for l in TempS .. mc-1 loop c.KeyEntry(l) := c.KeyEntry(l+1); end loop;
  TreePt.NumItems := M; c.NumItems := mc - 1; -- b => avail;
end if;
else -- b is page to left of TreePt.
  if TempS = 1 then
    b := c.P0;
  else
    b := c.KeyEntry(TempS-1).Pt;
  end if;
  mb := b.NumItems + 1; k := (mb-Mdiv2) / 2;
  if k > 0 then -- Move k items from page b to page TreePt.
    for l in reverse 1..(Mdiv2-1) loop TreePt.KeyEntry(l+k) := TreePt.KeyEntry(l); end loop;
    TreePt.KeyEntry(k) := c.KeyEntry(TempS); TreePt.KeyEntry(k).Pt := TreePt.P0;
    mb := mb - k;
    for l in reverse 1..(K-1) loop TreePt.KeyEntry(l) := b.KeyEntry(l + mb); end loop;
    TreePt.P0 := b.KeyEntry(mb).Pt;
    c.KeyEntry(TempS) := b.KeyEntry(mb); c.KeyEntry(TempS).Pt := TreePt;
    b.NumItems := mb-1; TreePt.NumItems := Mdiv2 - 1 + k; TooSmall := false;
  else -- Merge pages TreePt and b together.
    b.KeyEntry(mb) := c.KeyEntry(TempS); b.KeyEntry(mb).Pt := TreePt.P0;
    for l in 1..(Mdiv2-1) loop b.KeyEntry(l+mb) := TreePt.KeyEntry(l); end loop;
    b.NumItems := M; C.NumItems := mc - 1; -- TreePt => Avail
  end if;
end if;
end underflow;

```

procedure del(P: in out PagePt; TooSmall: in out Boolean) is

```

  TempPage: PagePt;
  -- TreePt and K are global.
begin
  TempPage := P.KeyEntry(P.NumItems).Pt;
  if TempPage /= null then
    del(TempPage, TooSmall);
    if TooSmall then
      underflow(p, TempPage, P.NumItems, TooSmall);
    end if;
  else
    p.KeyEntry(P.NumItems).Pt := TreePt.KeyEntry(k).Pt;
    TreePt.KeyEntry(k) := p.KeyEntry(P.NumItems);
    p.NumItems := P.NumItems - 1; TooSmall := (P.NumItems < Mdiv2);
  end if;
end del;

```

```

begin -- Start delete.
  if TreePt = null then
    put("Can not delete key, it is not in the tree."); new_line;
    TooSmall := false;
  else
    l := 1; r := TreePt.NumItems; -- Start binary search.
    loop
      K := (l+r) / 2;
      if DeleteKey <= TreePt.KeyEntry(k).Key then r := K - 1; end if;
      if DeleteKey >= TreePt.KeyEntry(k).Key then l := K + 1; end if;
      exit when l > r;
    end loop;
    if r = 0 then
      TempPage := TreePt.P0;
    else
      TempPage := TreePt.KeyEntry(r).Pt;
    end if;
    if (l-r) > 1 then
      if TempPage = null then
        TreePt.NumItems := TreePt.NumItems-1; TooSmall := (TreePt.NumItems < Mdiv2); --This
is a terminal page.
        for l in K..TreePt.NumItems loop TreePt.KeyEntry(l) := TreePt.KeyEntry(l+1); end loop;
      else
        del(TempPage,TooSmall);
        if TooSmall then
          underflow(TreePt,TempPage,r,TooSmall);
        end if;
      end if;
    else
      delete(DeleteKey,TempPage,TooSmall);
      if TooSmall then
        underflow(TreePt,TempPage,r,TooSmall);
      end if;
    end if;
  end if;
end delete;

-- Deletes the SearchKey from the tree if it exists.
procedure DeleteBtreeKey(SearchKey: in MyKey) is
  TempPage: PagePt;
  TooSmall: boolean;
begin
  delete(SearchKey, Root, TooSmall);
  if TooSmall then
    if Root.NumItems = 0 then
      TempPage := Root; -- Either the tree was empty or the old root split.
      Root := TempPage.P0; -- TempPage => Avail
    end if;
  end if;
end;
end BTree;

```

Not completely debugged

The following is a recursive algorithm. To locate the point of insertion, a procedure to do insertion must have at a minimum a pointer to the tree and the key to be inserted. If a child node splits, the child must report the fact to the parent (HasSplit), pass the middle key back to the parent (MiddleKey), and pass a pointer to the new node (NewNodePoint) back to the parent. For convenience, we assume that the minimum number of keys is even (M is odd) because it is easier to identify the middle key when a node splits. Assume Tree = Null to start.

```
procedure InsertKey(Tree: in out NodePointer; Key: in KeyType; HasSplit in out Boolean;
MiddleKey: in out KeyType; NewNodePoint: NodePointer) is
begin
```

```
    if Tree = null then insert first key in a new tree; return; end if;
    If the Tree is a leaf then
        If the number of keys < (M - 1) then
            Insert Key in current node in ascending key order;
            HasSplit := false;
        Else
            NewNodePoint <= Avail;
            Leave half keys in current node, move half ((M-1)/2) to node
                at NewNodePoint;
            Set MiddleKey to mid value of the keys in node that split;
            Set HasSplit := true;
            Insert(Tree, Key, HasSplit, MiddleKey, NewNodePoint);
        End If
    Else
        Locate pointer to next level in the tree (child), assign it to NextTree;
        InsertKey(NextTree, Key, Split, MiddleKey, NewNodePoint);
        If HasSplit then
            If the number of keys in tree < (M - 1) then
                Insert key and NewNodePointer in current node;
                HasSplit := false;
            Else
                NewNode <= Avail;
                Leave (M-1)/2 keys in Tree;
                Move (M-1)/2 keys and pointers to Node at NewNode
                MiddleKey := MiddleValue
                HasSplit := true;
                Insert(NextTree, Key, Split, MiddleKey, NewNodePoint);
                Tree := NewNodePoint;
            End if;
        End If;
    End If;
End;
```

```
procedure Insert(Tree in out Tree; Key: in KeyType; HasSplit: Boolean; MiddleKey: KeyType,
NewNodePoint: NodePointer) is
```

```
begin
    If HasSplit then
        NewTreeNode <= Avail; NewTreeNode.LeftMostChild := Tree;
        NewTreeNode.NextChild := NewNodePoint;
        NewTreeNode.FirstKey := MiddleKey; NewNodePoint := NewTreeNode;
    End if;
End;
```

--Based on an example by Dale and Walker in "Abstract Data," ISBN 0-669-40000-9

# Sorting

**Simple Bubble:** Comparative interchange.

| Pass 1 |    |    | Pass 2 |    | Pass 3 |
|--------|----|----|--------|----|--------|
| 9-     | 8  | 8  | 8-     | 3  | 3-     |
| 8-     | 9- | 3  | 3-     | 8- | 6-     |
| 3      | 3- | 9- | 6      | 6- | 8      |
| 6      | 6  | 6- | 9      | 9  | 9      |

Maximum of  $N-1$  passes. Can stop when a pass does not result in any exchanges. The Minimum compares is  $N-1$  if the data is in sorted order with no exchanges. If the data is in reverse sorted order: # compares = # exchanges :=  $(N-1) + (N-2) + (N-3) + \dots + 1 := N(N-1)/2$ . The average considering all permutations is  $N(N-1)/4$  exchanges. Hence the time  $T \propto N^2$  or  $O(N^2)$  as  $N$  becomes large. The algorithm takes advantage of natural order.

**True Bubble Sort:** Comparative interchange.

| Bubble 1 | Bubble 2 | Bubble 3 | Bubble 4 | Bubble 5 | Bubble 6 | Bubble 7 |      |
|----------|----------|----------|----------|----------|----------|----------|------|
| 19-      | 13 !     | 5        | 5 !      | 1        | 1        | 1        | 1    |
| 13-      | 19-!     | 13       | 13 !     | 5        | 5        | 5        | 5    |
| 5        | 5-       | 19-      | 19 !     | 13       | 13       | 13 !     | 13   |
| 27       | 27       | 27-      | 27-      | 19 !     | 19       | 19 !     | 16   |
| 1        | 1        | 1        | 1-       | 27-      | 26       | 26 !     | 19 ! |
| 26       | 26       | 26       | 26       | 26-      | 27-      | 27 !     | 26 ! |
| 31       | 31       | 31       | 31       | 31       | 31-      | 31-      | 27 ! |
| 16       | 16       | 16       | 16       | 16       | 16       | 16-      | 31-  |

The number of comparisons and exchanges depends on how badly the data is out of order. There are at most  $N-1$  more compares than exchanges. There are no exchanges if the data is in order and  $N-1$  compares. The maximum exchanges occurs when the list is in reverse order,  $N(N-1)/2$  compares and exchanges. The average considering all permutations is  $N(N-1)/4$  exchanges. Hence the time:  $T \propto N^2$ ,  $O(N^2)$ .

Best for lists:  $4 \leq N \leq 11$ . For  $N < 4$ , force with nested if statements.

The algorithm takes advantage of natural order. The code will run faster if you "bubble the hole" as opposed to complete exchanges.

## Bubbling the Hole

### Not All Programmers are Created Equal

As an example, assume we are comparing the keys at  $J = 5$  and  $(J-1) = 4$  with keys 7 and 27 respectively.

|   | Table prior to Exchange |   | Table after Exchange |  | Exchange Logic         |
|---|-------------------------|---|----------------------|--|------------------------|
| 1 | 5                       | 1 | 5                    |  |                        |
| 2 | 13                      | 2 | 7                    |  | Temp := Table[J]       |
| 3 | 19                      | 3 | 13                   |  | Table[J] := Table[J-1] |
| 4 | 27-                     | 4 | 19                   |  | Table[J-1] := Temp     |
| 5 | 7-                      | 5 | 27                   |  |                        |
| 6 | 26                      | 6 | 26                   |  |                        |
| 7 | 31                      | 7 | 31                   |  |                        |
| 8 | 16                      | 8 | 16                   |  |                        |

| Traditional Approach to Exchange                                                                                                                                                                    | Bubbling the Hole                                                                                                                                                                                                                                                                                                                                                                         |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> J := 5; While Table[J] &lt; Table[J-1] Loop   Temp := Table[J];   Table[J] := Table[J-1];   Table[J-1] := Temp   J := J - 1; End Loop </pre>                                                  | <pre> J := 5; Temp := Table[J]; While Temp &lt; Table[J-1] Loop   Table[J] := Table[J-1];   J := J - 1; End Loop Table[J] := Temp </pre>                                                                                                                                                                                                                                                  |
| <p>In the example above, 3 assignments are required for every exchange bubbling the data in the loop body. The total assignments above is 3 exchanges * 3 assignments/exchange = 9 assignments.</p> | <p>In the example above, 1 assignment is required for every exchange bubbling the data in the loop body plus 1 assignment for initialization and 1 final.</p> <p>The total assignments above is 3 in the body + 1 to set the hole (temp) to 7 initially + 1 final to move the hole temp into location 2 = 5 total exchanges. The greater the distance moved, the greater the savings.</p> |

## Sort by Selection: Interchange.

| Pass 1 | Pass 2 | Pass 3 | Pass 4 |   |
|--------|--------|--------|--------|---|
| 9-     | 5      | 5      | 5-     | 3 |
| 3      | 3      | 3      | 3-     | 5 |
| 8      | 8-     | 6--    | 6      | 6 |
| 6      | 6-     | 8      | 8      | 8 |
| 5-     | 9      | 9      | 9      | 9 |

On each pass, find the largest element and exchange it with the current bottom of the list. Decrease the list length by one for the next pass. Requires  $N-1$  passes. Does not take advantage of natural order. Note this requires a maximum of  $N-1$  exchanges and  $N(N-1)/2$  compares. Time:  $T \propto N^2$  or  $O(N^2)$ .

## Shell Sort: Comparative interchange

Strategy: Break long list into short list and sort using a bubble sort. Now combine these list and sort with bubble sort taking advantage of natural order. Time :=  $O(N^{1.5 + \epsilon/2})$ . There are many schemes for dividing the list. A commonly suggested one is:

- 1) Take  $d_1 := 2^k - 1$ , where  $2^k < N \leq 2^{k+1}$ .
- 2) For successive passes use  $d_{i+1} = \lceil (d_i - 1)/2 \rceil$ .

For example, if  $N = 1024$ , then  $2^9 = 512 \leq 1024 \leq 2^{10} = 1024$ . Hence  $d_1 := 2^9 - 1 = 512 - 1 = 511$ . Then  $d_2 := 255$ ,  $d_3 := 127$ ,  $d_4 = 63$ ,  $d_5 = 31$ ,  $d_6 = 15$ ,  $d_7 = 7$ ,  $d_8 = 3$ , and  $d_9 := 1$ . The method is known to run faster when  $d_{i+1}$  is chosen such that  $d_i / d_{i+1}$  is not an integer. That is each list contains some members from other lists.

Example using  $d_1 := 4$ ,  $d_2 := 2$ , and  $d_3 := 1$  for simplicity:

Pass 1

|     |     |     |    |
|-----|-----|-----|----|
| 13- | 01  | 01  | 01 |
| 19  | 19- | 19  | 19 |
| 21  | 21  | 21- | 16 |
| 26  | 26  | 26  | 26 |
| 01- | 13  | 13  | 13 |
| 31  | 31- | 31  | 31 |
| 16  | 16  | 16- | 21 |

Motivation: Sorting short list is linear opposed to longer list in exponential time. Combine list with overlap to take advantage of natural order in subsequent passes.

### Pass 2

|     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|
| 01- | 01  | 01  | 01- | 01  | 1   |
| 19  | 19- | 19  | 19  | 19  | 19  |
| 16- | 16  | 16- | 13- | 13  | 13  |
| 26  | 26- | 26  | 26  | 26- | 26  |
| 13  | 13  | 13- | 16  | 16  | 16- |
| 31  | 31  | 31  | 31  | 31- | 31  |
| 21  | 21  | 21  | 21  | 21  | 21- |

### Pass 3

|     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 01- | 01  | 01- | 01  | 01  | 01  | 01  | 01  | 01  | 01  | 01  | 01  |
| 19- | 19- | 13- | 13  | 13  | 13  | 13- | 13  | 13  | 13  | 13  | 13  |
| 13  | 13- | 19  | 19- | 19  | 19- | 16- | 16  | 16  | 16  | 16  | 16- |
| 26  | 26  | 26- | 26- | 26- | 16- | 19  | 19  | 19  | 19  | 19- | 19- |
| 16  | 16  | 16  | 16  | 16- | 26  | 26  | 26- | 26  | 26- | 21- | 21  |
| 31  | 31  | 31  | 31  | 31  | 31  | 31  | 31- | 31- | 21- | 26  | 26  |
| 21  | 21  | 21  | 21  | 21  | 21  | 21  | 21  | 21- | 31  | 31  | 31  |

**Quicksort:** Comparative interchange distributive sort.

Time:  $T \propto N \log_2 N$  to  $N^2$  (list already in sorted order).

Generally faster than shell sort or bubble sort if  $N \geq 12$ .

Storage approximately  $N + \log_2 N$ , bookkeeping space required for stack.

Simple:

- 1) Assume the top number in the list estimates the median. Split list into two sub list based on this estimate of the median. Search from the top looking for a number larger than the median estimate and from the bottom looking for a number smaller than the median estimate. Exchange these numbers. Repeat. When the scan from the top meets the scan from the bottom, exchange the element at this position with the median estimate dividing the list.
- 2) Now sort the sub lists in the same manner. Save the address of the sub lists in a stack. Always sort the shortest sub list first.

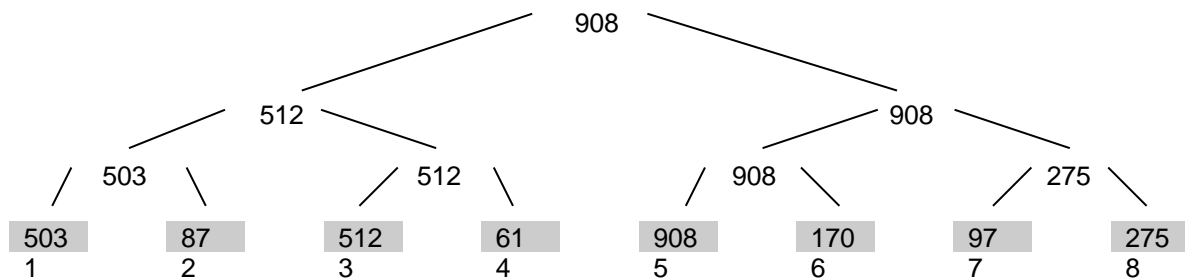
A perfect choice for the median on each list reduces the space required for the stack as well as the run time. This results in about a **34% reduction in runtime**. Suggestion, compare the top item in the sub list, the last item in the sub list, and the item from the center of the sub list. **Use the median of these three numbers as the estimated median of the sub list prior to starting the sort for the sub list.**



When the size of a list drops to 12 or below, use a bubble sort on the sub list. This also makes it run faster. If the list size is less than 4, use a nested "If" to complete the sort. Does not take advantage of natural order.

| # 1 |     | # 2 |    | #3 |     | # 3 |     |    |
|-----|-----|-----|----|----|-----|-----|-----|----|
| 13  | 13  | 12  | 12 | 01 | 01  | 01  | 01  | 01 |
| 19- | 01  | 01  | 01 | 12 | 12  | 12  | 12  | 12 |
| 21  | 21- | 13  | 13 | 13 | 13  | 13  | 13  | 13 |
| 12  | 12- | 21  | 21 | 21 | 21  | 21  | 19  | 16 |
| 31  | 31  | 31  | 31 | 31 | 31- | 16- | 16- | 19 |
| 01- | 19  | 19  | 19 | 19 | 19  | 19- | 21  | 21 |
| 16  | 16  | 16  | 16 | 16 | 16- | 31  | 31  | 31 |

## Heap Sort:



The tree structure should really only store pointers. This algorithm is efficient for large N. Space = N + (N-1), i.e., N for the data and N-1 for the tree.

Assume N = 1000, approximate run times:

160,000 units for heapsort -- data independent  $O(N \cdot \log_2 N)$ , guaranteed run time

130,000 units for shell -- data dependent  $O(N^{1.5} \text{ to } N^2)$

80,000 units for quicksort -- data dependent  $O(N \cdot \log_2 N \text{ to } N^2)$

Bubble Sort  $O(N \text{ to } N^2)$

As N gets larger, heapsort gets better than shell sort. It never gets better than quicksort if the data is truly random.

## Algorithm Heapsort: by J. W. J. Williams [CACM 7(1964), 347-348]

Assume records  $R_1, \dots, R_N$ . The records are rearranged in place, i.e., their keys are ordered such that  $K_1 \leq K_2 \leq \dots \leq K_N$ . First we rearrange the records to form a heap, then we repeatedly remove the top of the heap and transfer it to its final position. We assume that  $N \geq 2$ . The heap is initially built in the 1 to  $r$  spaces holding records to be sorted.

Set  $m := \lfloor N/2 \rfloor + 1$ ;  $r := N$ ;

MainLoop;  
loop

```

    if (m > 1) then --True implies we are forming the initial heap.
        m := m - 1; R := Rm; K := Km;
    else -- The keys K1, K2, ..., Kr constitute a heap.
        R := Rr; K := Kr; Rr := R1; r := r - 1;
        if (r = 1) then --This terminates the algorithm.
            R1 := R; exit MainLoop;
        end if;
    end if;
end if;

```

j := m; --Prepare for shift up.

AdvanceDownward:

loop

```

    i := j; j := 2j; --Advance downward.

    if (j < r) then
        if (Kj < Kj+1) then --find "larger son"
            j := j + 1;
        end if;
        if (K >= Kj) then --larger than K?
            Ri := R; --Store it.
            exit AdvanceDownward;
        else
            Ri := Rj; --Move it up.
        end if;

    elsif (j = r) then -- Larger than K?
        if (K >= Kj) then -- Larger than K?
            Ri := R; -- Store it.
            exit AdvanceDownward;
        else
            Ri := Rj; -- Move it up.
        end if;

    elsif (j > r) then
        Ri := R; -- Store it.
        exit AdvanceDownward
    end if;

```

end loop AdvanceDownward;

end loop MainLoop;

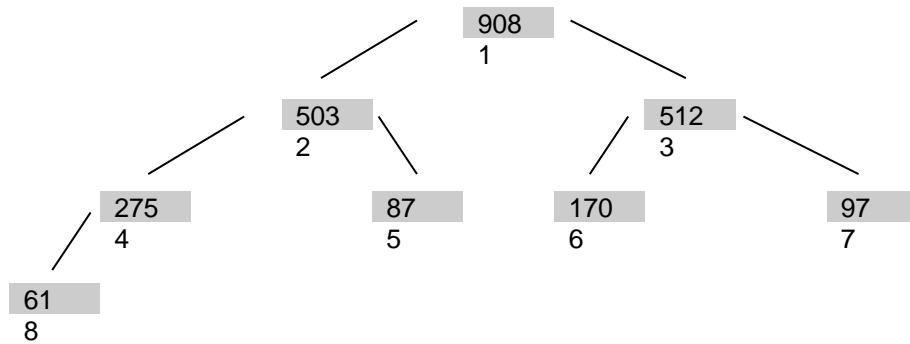
The heap creation phase is when  $r = N$  and  $m$  decreases to 1. The selection phase is when  $m = 1$  and  $r$  decreases to 1.

Record

|     |     |              |
|-----|-----|--------------|
| 1   | key | non key data |
| 2   |     |              |
| 3   |     |              |
| 4   |     |              |
| ... |     |              |
| N-2 |     |              |
| N-1 |     |              |
| N   |     |              |

| k1  | k2  | k3  | k4  | k5  | k6  | k7 | k8  |  | m | r | K   |
|-----|-----|-----|-----|-----|-----|----|-----|--|---|---|-----|
| 503 | 87  | 512 | 61  | 908 | 170 | 97 | 275 |  | 4 | 8 | 61  |
| 503 | 87  | 512 | 275 | 908 | 170 | 97 | 61  |  | 3 | 8 | 512 |
| 503 | 87  | 512 | 275 | 908 | 170 | 97 | 61  |  | 2 | 8 | 87  |
| 503 | 908 | 512 | 275 | 87  | 170 | 97 | 61  |  | 1 | 8 | 503 |
| 908 | 503 | 512 | 275 | 87  | 170 | 97 | 61  |  | 1 | 7 | 61  |

At this point the heap is built and we are ready to start the selection process.



Heap ready for first output.

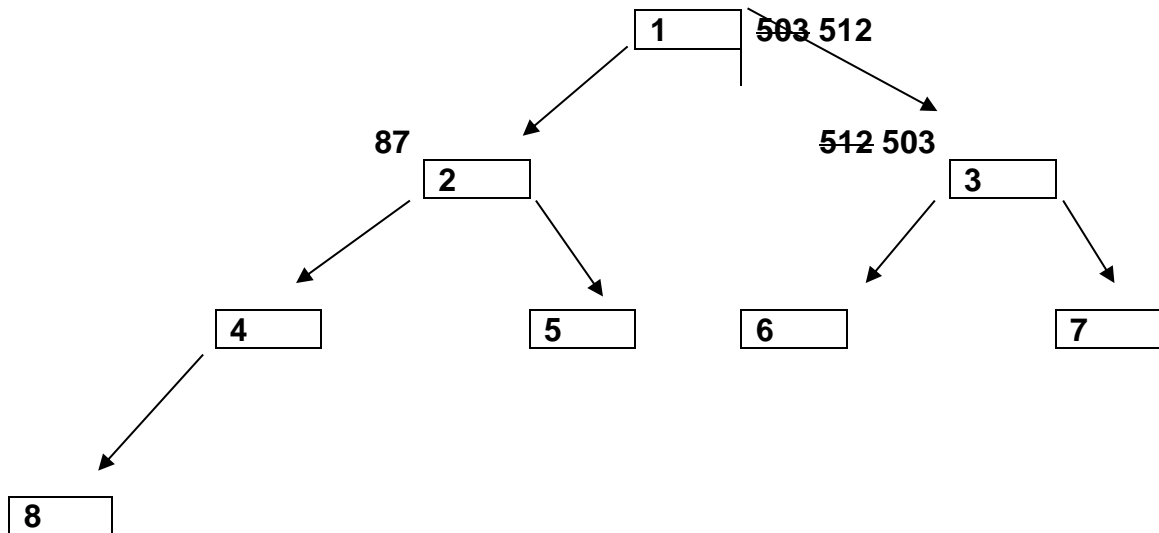
| k1  | k2  | k3  | k4  | k5  | k6  | k7  | k8  |  | m | r | K  |
|-----|-----|-----|-----|-----|-----|-----|-----|--|---|---|----|
| 512 | 503 | 170 | 275 | 87  | 61  | 97  | 908 |  | 1 | 6 | 97 |
| 503 | 275 | 170 | 97  | 87  | 61  | 512 | 908 |  | 1 | 5 | 61 |
| 275 | 97  | 170 | 61  | 87  | 503 | 512 | 908 |  | 1 | 4 | 87 |
| 170 | 97  | 87  | 61  | 275 | 503 | 512 | 908 |  | 1 | 3 | 61 |
| 97  | 61  | 87  | 170 | 275 | 503 | 512 | 908 |  | 1 | 2 | 87 |
| 87  | 61  | 97  | 170 | 275 | 503 | 512 | 908 |  | 1 | 1 | 97 |
| 61  | 87  | 97  | 170 | 275 | 503 | 512 | 908 |  |   |   |    |

**Self Balancing Binary Trees:** also known as **AVL Trees** and **Height Balance Trees**. See any standard text on data structures.

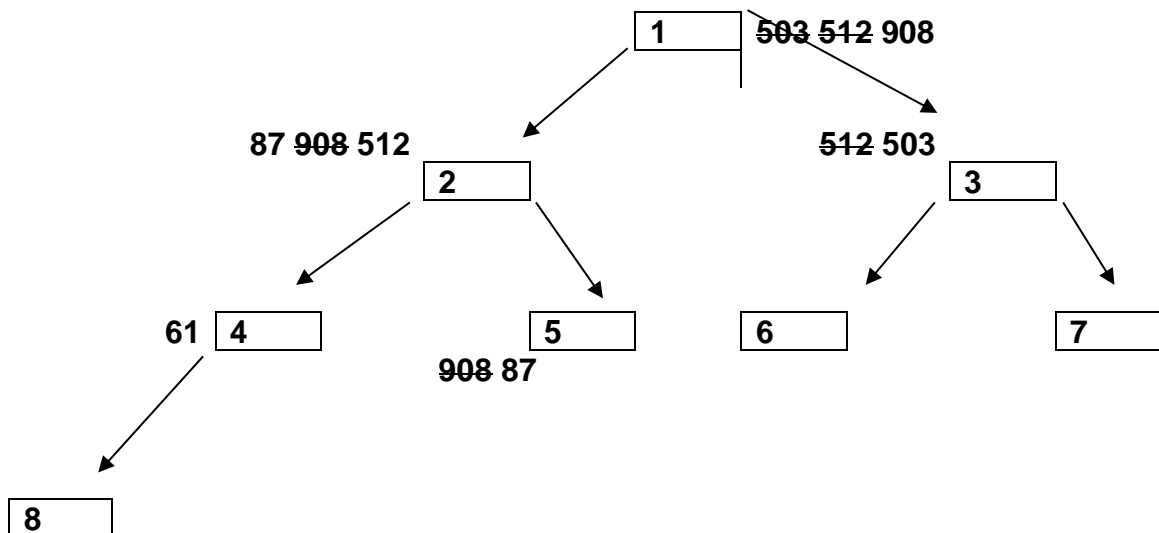
Tree after building initial heap using the following data: 503, 87, 512, 61, 908, 170, 97, and 275. Remember, we define a heap as a binary tree where the contents of the parent is always larger than or equal to its children content.

Note the children of node K are always at  $2K$  and  $2K + 1$ . The parent of node K is always at the floor of  $K/2$ . The tree is stored in a sequentially allocated array.

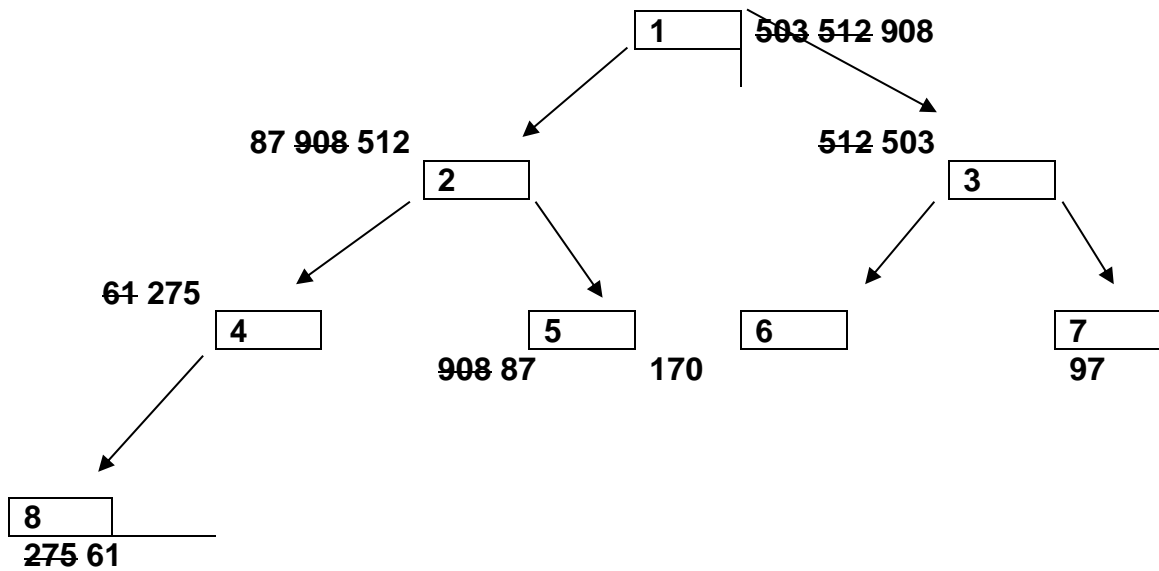
Heap after adding 512,  $N = 3$  items.



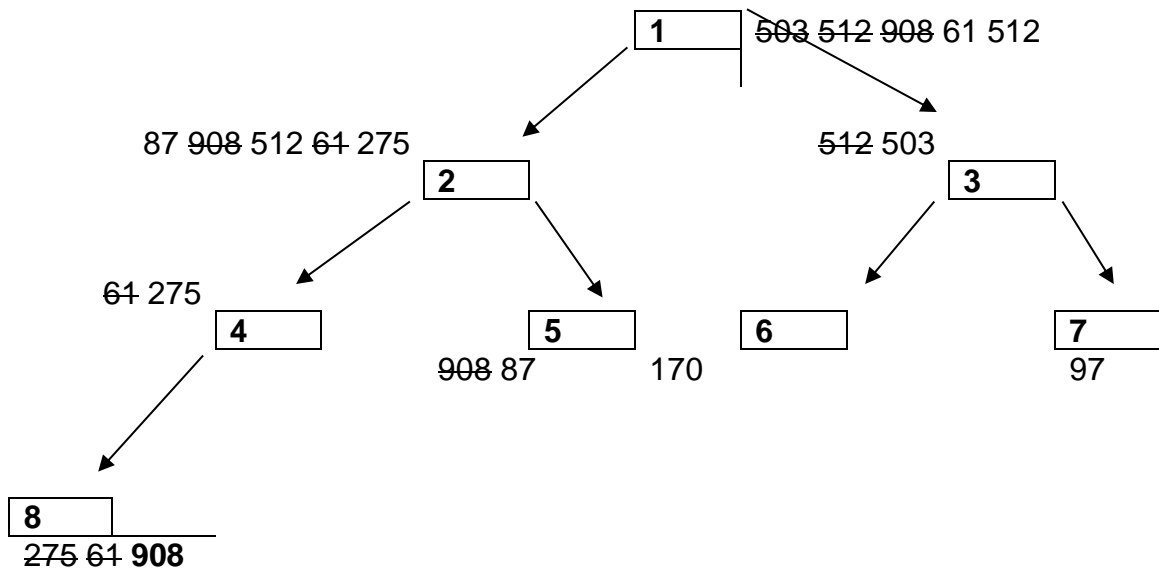
Heap after adding 908,  $N = 5$



Final heap prior to start of sort N = 8.



Heap after to first item placed in sorted order of sort N = 7.



Note the parent of the node at location K is at the lowerbound of  $K/2$ . The children of the node at location K are at  $2K$  and  $2K + 1$ .

# Hash Sort

Assume the hash address =  $\lceil \text{key}/2 \rceil$ , e.g. hash address :=  $\lceil 19/2 \rceil := \lceil 9.5 \rceil = 10$ . Note that a perfect or nearly perfect hash function would have time proportional to  $N$ . If all keys hash to the same address, the time will be proportional to  $N^2$ . The distribution pass (hash) is usually followed by another pass to place the records in consecutive locations. Older literature will refer to this as a “scatter write” followed by a “gather read.”

| keys | loc |    | collisions |
|------|-----|----|------------|
| 19   | 1   | 3  |            |
| 13   | 2   | 6  | ↑          |
| 8    | 3   | 7  |            |
| 27   | 4   | 8  | ↑          |
| 3    | 5   |    |            |
| 14   | 6   |    |            |
| 7    | 7   | 13 | ↓          |
| 16   | 8   | 14 | ↓          |
| 17   | 9   | 16 | ↓          |
| 6    | 10  | 17 | ↓          |
| 24   | 11  | 19 |            |
|      | 12  | 24 |            |
|      | 13  |    |            |
|      | 14  | 27 |            |
|      | 15  |    |            |
|      | 16  |    |            |

The hash pass has traditionally been referred to as a “Scatter Write.” This is followed by a second pass to consolidate the table into contiguous entries called a “Gather Read.”

## Radix Sorting:

| Start |   | Bucket         |  | Merge |   | Bucket         |  | Merge |
|-------|---|----------------|--|-------|---|----------------|--|-------|
| 19    | 0 |                |  | 01    | 0 | 01, 02, 05, 09 |  | 01    |
| 13    | 1 | 01, 31, 11, 21 |  | 31    | 1 | 11, 13, 16, 19 |  | 02    |
| 05    | 2 | 02             |  | 11    | 2 | 21, 26, 27     |  | 05    |
| 27    | 3 | 13             |  | 21    | 3 | 31             |  | 09    |
| 01    | 4 |                |  | 02    | 4 |                |  | 11    |
| 26    | 5 | 05             |  | 13    | 5 |                |  | 13    |
| 31    | 6 | 26, 16         |  | 05    | 6 |                |  | 16    |
| 16    | 7 | 27             |  | 26    | 7 |                |  | 19    |
| 02    | 8 |                |  | 16    | 8 |                |  | 21    |
| 09    | 9 | 19, 09         |  | 27    | 9 |                |  | 26    |
| 11    |   |                |  | 19    |   |                |  | 27    |
| 21    |   |                |  | 09    |   |                |  | 31    |

**Each pass of radix is performed from least to most significant digit/character.**

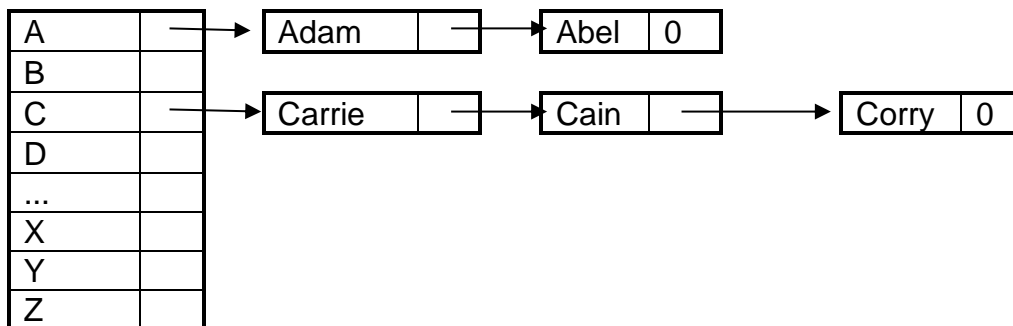
Note only two passes regardless of the number of two digit keys.

Note the number of buckets depends on the range of the key.

Efficient to implement as linked lists:

List

Head



Approximate Space:

- 1) Array for buckets: (number buckets) \* (number records)\*(space 1 record)
- 2) Linked list: (number buckets) + (number records)\*(space 1 record + link)

Time will depend on the number of passes (one pass for each character in the key) and the number of records. The algorithm should be very fast. This algorithm works well with duplicate records retaining them in the same order after sorting as they appeared in the raw data, a stable sort. **Consider 1000 five character (alphabetic keys). Bubble sort could and sort by selection would take 999 passes. Radix sort would complete in 5 passes!**

## Straight Insertion:

This algorithm sorts records  $R_1, \dots, R_N$  in place with their keys in ascending order.

```
for J in 2..N loop
  I := J - 1; K := KJ; R := RJ;

  MoveLoop:
  loop
    if K >= KI then
      RI+1 := R; -- This is the final position.
      exit MoveLoop;
    else
      RI+1 := RI; --Search higher.
      I := I - 1;
      if I = 0 then
        R1 := R;
        exit MoveLoop;
      end if;
    end if;
  end loop MoveLoop;
end loop;
```

Record

|     | key | non key data |
|-----|-----|--------------|
| 1   | 45  | Joe --       |
| 2   | 23  | Sam --       |
| 3   | 17  | Mary --      |
| ... | 65  | Bob --       |
| N-2 | 57  | Sara --      |
| N-1 |     |              |
| N   |     |              |

Note the worst case number of compares (and record moves) is  $1 + 2 + 3 + \dots + (N-2) + (N-1) = N(N-1)/2$ . Hence the time is proportional to  $N^2$ . The worst case occurs when the records are in reverse sorted order. The best case occurs when the records are in sorted order implying  $N-1$  compares and zero moves. Hence the algorithm prefers data nearly in sorted order or records very close to their final sorted position. For all permutations, the expected value for the number of moves and compares should be closer to  $N(N-1)/4$  which is still proportional to  $N^2$ , i.e.,  $O(N^2)$ . This algorithm bubbles the hole for efficiency.



## Sorting by insertion:

Assume a table that is normally maintained in random order such as a hash table. Assume that we must also be able to process the data in ascending key order. We desire that one application be able to search the hash table at the same time we are sorting it. Another example would be an activity oriented list. The list is ordered in decending order on frequency of use (random order by key) and searched sequentially. This will minimize search time when most accesses are to the same table elements. Again, we wish to be able to travers the list in sorted order but do not wish to disturb the order required by other applications. The following algorithm will meet these requirements. For convenience, the table is assumed to be full. The extension to a table that is not full is trivial.

### Algorithm Linked Insertion Sort:

| Location | logical record | key field | information field | link field (sorted) | I = 3 | I = 2 | I = 1 |
|----------|----------------|-----------|-------------------|---------------------|-------|-------|-------|
| 0)       | R <sub>0</sub> | dummy     | dummy             | L <sub>0</sub> = 2  | 4     | 4 2   | 2     |
| 1)       | R <sub>1</sub> | 182       | info              | 0                   |       |       | 0     |
| 2)       | R <sub>2</sub> | 17        | info              | 4                   |       | 4     | 4     |
| 3)       | R <sub>3</sub> | 84        | info              | 1                   | 0     | 0     | 0 1   |
| 4)       | R <sub>4</sub> | 27        | info              | 3                   | 0 3   | 3     | 3     |

Assume the table of N items is store in the above format. Location zero is a dummy entry to make sorting convenient. L<sub>0</sub>, the link field of record 0, is used as a list head to the linked list of sorted records. N is the number of entries currently in the table (4 in the example above). The link fields are shown after the pass for I = 3, 2, and 1. Values changed in the pass are reduced in size and striked. For sorted data the number of compares for N records is N – 1. If the data is in reverse order the number of compares is N \* (N-1) / 2, i.e., O(N<sup>2</sup>).

### To Sort:

```

Set L0 := N, LN := 0
For I = (N-1), (N-2), ..., 1 loop
    P := L0, Q := 0, Key := KeyI

    While (P <> 0) and (Key > KeyP) loop
        Q := P; P := LQ;
    end loop;

    LQ := I; LI := P;
end loop;

```

### To Traverse:

```

Set P := L0;
While P <> 0 loop
    Visit the node pointed to by P;
    P := LP
end loop;

```

## Sorting Techniques

### 1) Significant factors:

- A) The number of records which can be held in main memory.
- B) The number of comparisons and the amount of information retained.
- c) The number of exchanges which must be made.
- D) Ability to take advantage of natural order in the data or preference for truly random order.
- E) Requirement for efficient use of storage.
- F) The number of records to be sorted.
- G) The length of the keys is “comparison” is utilized.
- H) Availability of auxiliary storage and requirements for efficient use.
- I) Difficulty to implement.
- J) Stability (are records with the same primary key and / or secondary key value in the same relative position to each other after the sort?).

### 2) To minimize sort time, we must in general reduce the number of moves and / or the number of compares required to place the records in sorted order. In many cases a sort can be improved by sort pointers (indexes) to the actual records rather than moving the actual records.

|   | pointer |
|---|---------|
| 1 | 2       |
| 2 | 5       |
| 3 | 1       |
| 4 | 4       |
| 5 | 3       |

|   | key    | non-key |
|---|--------|---------|
| 1 | Mary   | other   |
| 2 | Bob    | other   |
| 3 | Tom    | other   |
| 4 | Sara   | other   |
| 5 | Martha | other   |

**3) Basic types of sorts:**

A) Minimal memory requirements:

- a) comparison
- b) selection
- c) exchange

B) Available memory greater than space required for records.

- a) computation
- b) distribution

SortMM.doc

# Hashing

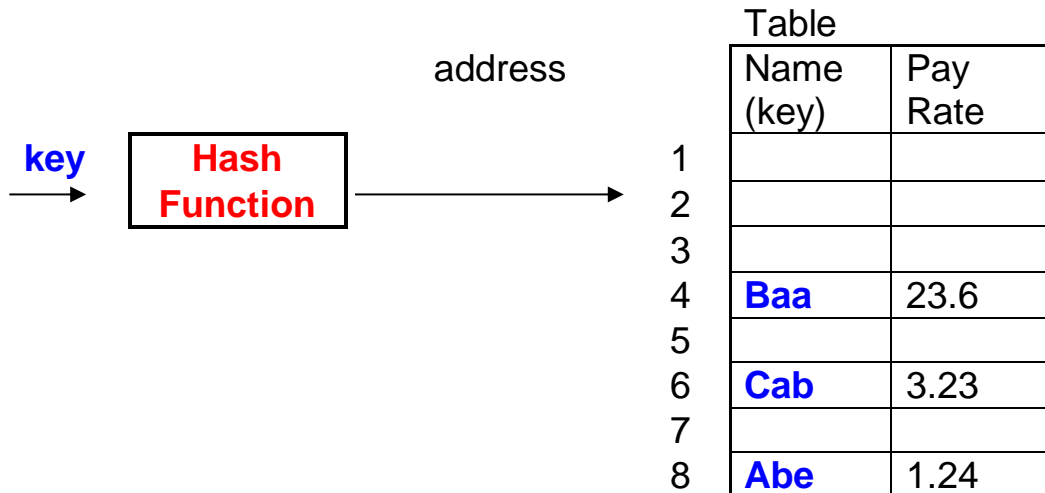
When searching a table we generally wish to minimize the number of probes required to locate a key or determine that the key is not in the table. Ideally, we would like to locate a key or determine it is not in the table in a single probe. We might also like to efficiently insert new keys and delete existing keys in addition to a fast search capability.

## Traditional Search Methods assuming N keys in a table of size N.

|                                       | Min. (find)                                           | Max. (find)                                                                  | Avg. (find)                                               |
|---------------------------------------|-------------------------------------------------------|------------------------------------------------------------------------------|-----------------------------------------------------------|
| Linear (random order)                 | 1                                                     | N                                                                            | $(N+1)/2$                                                 |
| Linear (sorted)                       | 1                                                     | N                                                                            | $< (N+1)/2$                                               |
| Binary Search (Sequential Allocation) | 1                                                     | $\log_2 N$                                                                   | Approximately $\log_2 N$                                  |
| Fibonacci                             | 1                                                     | Slightly better than a binary search                                         | Slightly better than a binary search                      |
| Binary or Alphabetic Search Tree      | 1                                                     | $\log_2 N$ to N depending on balance                                         | $\log_2 N$ to N depending on balance                      |
| B-Tree of order m                     | Get 1 <sup>st</sup> level of tree, + 1 probe in level | Level $\leq 1 + \log_{\lceil m/2 \rceil} (N+1)/2$ plus probes on each level. |                                                           |
| Trie                                  |                                                       |                                                                              |                                                           |
| Interpolation Search                  | 1                                                     |                                                                              | $\log_2 \log_2 N$                                         |
| Hash                                  | 1                                                     | 1 for perfect hash function to N if all keys collide                         | Depends on hash function and collision handling technique |
| Activity Oriented List                | 1                                                     | N                                                                            | W. P. Hsing 80 / 20 rule                                  |

In hashing, a unique key representing the object to be stored is presented to a “hash function.” The hash function massages the key forcing it to yield the address it wishes to be placed in the hash table. Anytime in the future the object is desired, its key is again presented to the hash function to determine where the object was previously stored. A good hash function normally distributes random keys uniformly across the table.

# Hashing



Each record stored in the hash table is associated with a unique identifier called the record “key.” Consider the following example. Assume the hash address is found by summing the numeric values associated with the letters in the key, i.e., the letters A..Z are mapped into the numbers 1..26.

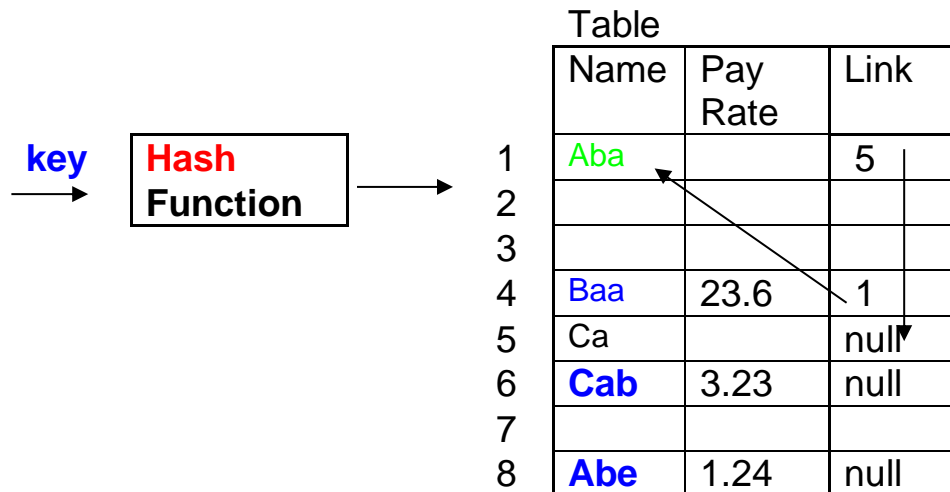
|        |        |        |        |
|--------|--------|--------|--------|
| A = 1  | B = 2  | C = 3  | D = 4  |
| E = 5  | F = 6  | G = 7  | H = 9  |
| ...    |        |        |        |
| W = 23 | X = 24 | Y = 25 | Z = 26 |

**Abe**  $\Rightarrow 1 + 2 + 5 = 8$

**Cab**  $\Rightarrow 3 + 1 + 2 = 6$

**Baa**  $\Rightarrow 2 + 1 + 1 = 4$

**Aba**  $\Rightarrow 1 + 2 + 1 = 4$  resulting in a collision.



Assume the hash address is found by summing the values associated with the letters in the key.

|        |        |        |        |
|--------|--------|--------|--------|
| A = 1  | B = 2  | C = 3  | D = 4  |
| E = 5  | F = 6  | G = 7  | H = 9  |
| ...    |        |        |        |
| W = 23 | X = 24 | Y = 25 | Z = 26 |

**Baa** =>  $2 + 1 + 1 = 4$

**Aba** =>  $1 + 2 + 1 = 4$  the collision handler selects location 1.

**Ca** =>  $3 + 1 + 1 = 5$  the collision handler selects location 5.

When a collision occurs, another location must be found for the record. Traditional collision handling techniques include linear probe and random probe. Java uses direct chaining (a linked list) to associate records that have collided after the collision handling technique has found a storage location. The next time "Aba" is accessed, the collision will occur again. In the future Java will search the collision chain to locate the record hopefully minimizing intermediate collisions. Note: Baa will always require 1 probe to locate, Aba two probes, and Ca three probes.

# Characteristics of a Good Hash Function

- 1) A good hash function is fast. It does not require a lot of computations or table lookups.
- 2) A good hash function randomizes the calculated addresses uniformly over the available set of addresses (uniform distribution).
- 3) An ideal hash function randomizes each key to a unique address.

Idea hash functions are generally hard to create and seldom worth the effort. In practice a reasonable number of collisions do not adversely impact performance. The reason that ideal hash functions do not occur without significant effort is due to the Laws of Probability. As an example consider the Saint Petersburg Paradox, sometimes called the birthday party paradox.

Paraphrased, the Saint Petersburg Paradox states that given 23 or more people there is a high probability that two or more of them will have the same day and month of birth. In fact, given 23 people, the probability that no two people will have the same month and day of birth is only 0.4927%, less than half. Two people with the same month and day of birth may be thought of as a collision.

The number of collisions may frequently be reduced by taking advantage of known key properties when creating the hash function. If the file is to be kept in auxiliary storage, then disk access time (milliseconds) when searching for a record is more important than CPU time (nano or pico seconds) to create the hash address. Hence it is frequently desirable to spend more time on the hash function to reduce collisions and the resulting disk accesses in auxiliary storage. For main memory tables, the time required to make a probe using a collision handling technique is not as significant.

## Generating Hash Addresses

- 1) **Direct address:** Use the key as the hash address. For example use the last four digits of a social security number as the hash address. Note that this requires a table of 10,000 entries (0-9999) even if there are only 200 employees.

It is possible to treat alphanumeric data also as a key. Consider a three character key. In binary this would correspond to 24 bits (8 per character or 1 byte). Hence the required address space would be  $2^{24} = 16,777,216$  locations. The large address space is prohibitive.

- 2) **Simple Codes:** Treat each letter of the alphabet as a number, i.e., A = 1, B = 2, C = 3, D = 4, E = 5, ..., Z = 26. Hence ABE would hash to  $1 + 2 + 5 = 8$ . Hence EBA =  $5 + 2 + 1 = 8$  would result in a collision as would any

permutation of these characters. The saving grace is that most English words are not an exact permutation (transpositions) of the same letters.

- 3) **Weighted Codes:** Most of the collisions caused by simple codes can be avoided by using weighted codes. For example, multiply each character by its position in the string from right to left. Hence ABE hashes to  $1*1 + 2*2 + 3*5 = 20$ . EBA hashes to  $1*5 + 2*2 + 3*1 = 12$ . This would eliminate the collision.
- 4) **Square and Extract N Bits:** Assume a table of size  $2^n$ . Then any address in the table (0 through  $(2^n-1)$ ) may be represented using  $n$  bits in binary. The general technique is to treat the key as a binary number (even if it is alphanumeric). The key is squared and  $n$  bits from the product extracted to form the hash address. Note that the product of  $n$  binary bits by  $n$  binary bits may produce a result up to  $2n$  bits in length. Hence the selection of the  $n$  bits for the hash address is important. Consider the following two examples.

**Case 1:** Consider four bit integers. We need a hash table of size  $2^{*4} = 16$  locations. Four bits from the square of the key may be used to address any location in the hash table. As an example use 3 base 10 = 0011 base two. Then  $3 * 3 = 9$  base 10 or 0001001 base 2.

$$\begin{array}{cccccccc}
 & & & & & 0 & 0 & 1 & 1 \\
 & & & & & * & 0 & 0 & 1 & 1 \\
 & & & & & \hline
 & & & & 0 & 0 & 0 & 1 & 1 \\
 & & & 0 & 0 & 0 & 1 & 1 & & \\
 & & 0 & 0 & 0 & 0 & 0 & & & \\
 & 0 & 0 & 0 & 0 & 0 & & & & \\
 \hline
 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & & 
 \end{array}$$

Note that for integers, if you select the left most  $n$  bits, the hash addresses will tend to zero as the leading bits of integers will be zeros. Put another way, we are using the high order bits of the number to generate the key and ignoring the low order bits. For this example we would use 0000 as the hash address (left most four bits). Note that 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, and 15 will all generate the hash address of 0 using the left most 4 bits of the product.

If you use the right most 4 bits, 1001 in this case, the hash address is most dependent on the low order bits of the key and the high order bits are ignored. The best hash address would probably be using the middle four bits in the product, 0010 in this case, as the middle bits are dependent on all the bits of the key.



- Case 2:** Note that alphabetic fields are normally left justified in character fields and padded with spaces on the right (as opposed to right justification for integers and padded on the left with zeros). Hence using the left most n bits from the product of alphabet fields would tend to ignore the right most characters of the string. Using the right most n bits would tend to generate collision base on the value of the space character squared. Again, the best address will most likely come from the middle n bits of the product as they depend on all characters in the key.
- 5) **Folding:** Folding is any technique used to take a long key and combine its parts into a smaller more manageable size. For example assume a 32 bit integer machine using 8 bit ASCII characters. Then an 8 character string would consume 64 bits or 8 bytes. The 8 character string could be treated as two 4 character strings of 32 bits each and added. The sum would be 32 bits. Any overflow is normally ignored. The sum would be dependent on the whole key. Since it may be treated as an integer, it can then be further manipulated to form a hash address. Folding is normally used to reduce the size of a key that exceeds the limits of an integer on the target machine to a bit string that can be represented as an integer. Note that adding might produce a negative number on two's complement machines. The use of the logical "or" operation for folding tends to produce all 1's. The use of the logical "and" operator tends to produce all 0's. The logical "exclusive or" operator on the other hand tends to randomize the pattern of 0's and 1's produced and hence is frequently used for folding.
- 6) **Division Remainder:** Assume a table with TS entries. The hash address HA is the remainder from dividing the key by the TS, or  $HA = \text{key} \text{ Mod } TS$ . Note that  $0 \leq HA < TS$ . The division remainder technique is good as a general hash function when the characteristics of the keys are not know ahead of time. It also has the advantage that the table may be of arbitrary size.

In general, the divisor (table size) should be a prime number or at least an odd number that is not evenly divisible by small odd numbers such as 3, 5, 7, etceteras. For example, consider a table of 1000 entries. We will use base 10 arithmetic for convenience. Assume we wish to hash 800 six digit even integers into this table. Note that if the  $HA = \text{key} \text{ Mod } TS$ , then all 800 keys will hash to even addresses causing at least 300 collisions as there are only 500 even addresses. Note that  $1000 = 10^{**3}$ . Further note that any key / 1000 will produce a quotient and remainder consisting of the last three digits of the key unmolested, e.g., the remainder of  $5436 / 1000$  is 436. Hence the hash address will always be the last three digits of the key. Since all keys were assumed to be even numbers, all 800 hash addresses will be even numbers. Theoretically, prime numbers do the best job of scrambling the key to produce uniformly distributed remainders

(hash addresses). **RULE:** Never use a tables size that is a power of the numeric base.

For base 2, note that a table of size  $256 = 2^{**}8$ . Hence the HA = key mod 256 will be the last eight bits of the key unmolested. On a binary computer, in general, the table size should not be a power of 2 since the remainder will always be the last n bits of the key. If  $TS = 2^{**}n$ , then  $HA = \text{key mod } TS$  produces a remainder consisting of the last n bits in the key.

Alphabetic keys represent a special case. ASCII letters transpose as multiples of 3. For example, in ASCII A = 65, B = 66, and C = 67. implying AB = 6566 and BA = 6665. BA – AB = 99, and  $99 / 3 = 33$  a multiple of three. CA – AC = 6765 – 6567 = 198, and  $198 / 3 = 66$ , a multiple of 3. Hence the TS for alphabetic keys should not be a multiple of 3 or excessive collisions may be expected.

### A Special Case:

Normally a good hash function takes a key and randomizes it uniformly across the address space. There are exceptions. Assume a four digit numbering system for purchases orders, i.e., 0000 through 9999. Assume that purchase orders are issued in consecutive order, we pay the oldest purchase orders that have been returned with the merchandise at the end of each month (smallest PO numbers), and that we start over with PO number 0000 if we issue all PO numbers. Further assume that we issue a maximum of 10,000 purchase orders per year but that there have never been more than 1,000 outstanding purchase orders at any given time in the system. Because of this we would like to limit the size of the hash table to 1,000 entries rather than 10,000. Assume that the hash table contains address 0 – 9999. We will compute the hash address as the remainder of dividing the PO number by 1000. Sample calculations follow.

| PO#  | Hash Address                        |      | Hash Table |
|------|-------------------------------------|------|------------|
|      |                                     | 0    |            |
|      |                                     | 1    |            |
|      |                                     | 2    |            |
|      |                                     | 3    |            |
|      |                                     | ∞    |            |
| 1768 | $HA = 1768 \text{ mod } 1000 = 768$ | 1768 | PO 1768    |
| 1769 | $HA = 1769 \text{ mod } 1000 = 769$ | 1769 | PO1769     |
| 1770 | $HA = 1770 \text{ mod } 1000 = 770$ | 1770 | PO1770     |
|      |                                     | ∞    |            |
|      |                                     | 9999 |            |

Note that a collision will occur for PO 2768 as the Ha =  $2768 \text{ mod } 1000 = 768$ , the same as the HA for PO 1768. The probability of an actual collision however is very small due to our policy of paying the oldest outstanding PO's first. By the time the goods for PO 2768 arrive, PO 1768 should already have been received and paid freeing that slot in the table.

```
-- Sample code in Ada showing how to treat text strings
-- as integers to generate hash addresses.
```

```
--file UnCheck1.adb
with Ada.Unchecked_Conversion, Ada.Text_IO; use Ada.Text_IO;
```

**procedure UnCheck1 is**

```
package LongIntIO is new Ada.Text_IO.Integer_IO(Long_Integer);
use LongIntIO;
```

```
type String4 is new String(1..4);
```

```
function ConvertString4 is
  new Ada.Unchecked_Conversion(String4, Long_Integer);
```

```
A: String4 := "ABCD";  
B: String4 := "EFGH";
```

```
Sum: Long_Integer;
begin
  Sum := ConvertString4( A )/2 + ConvertString4( B )/2; //Folding
        //Dividing by a prime (3, 5) may produce better results.
  put("The sum of ""ABCD"" + ""DEFG"" is "); put(Sum);
end;
```

[illegible]

The sum of "ABCD" + "DEFG" is **1178944578**

**NOTE:** Its better to instruct the compiler to ignore integer overflow. Then you can use the entire number rather than dividing by 2. Division by 2.15 or another small fraction may be better. If integer overflow is ignored, ABS(Sum) would be more appropriate to compensate for the sign bit.

### Convenient trick termed “slice” in ADA:

**B[1..2] := A[3..4]; -- sets first 2 characters of B to last 2 of A.**

```
Sum := ConvertString4( A[1..4] ) / 3; -- 1..4 is a “slice” or substring.
```

```

--file UnCheck2.adb  **** Compiler instructed to ignore "integer overflow"
with Ada.Unchecked_Conversion, Ada.Text_IO; use Ada.Text_IO;
procedure UnCheck2 is
  package LongIntIO is new
    Ada.Text_IO.Integer_IO(Standard.Long_Integer);
  use LongIntIO;

  type String4 is new String(1..4);
  function ConvertString4 is new
    Ada.Unchecked_Conversion(String4, Long_Integer);

  A: String4 := "ABCD";
  B: String4 := "EFGH";
  Sum, HA: Long_Integer;
begin  -- first fold strings to single integer 32 bit integer
  Sum := ConvertString4(A) + ConvertString4(B); -- overflow off

  put("The sum of ""ABCD"" + ""DEFG"" is "); put(Sum); new_line;
  -- Division remainder with folding and table size = 256: addresses from 1 thru 256
  HA := abs(Sum) mod 256 + 1; -- for a table with locations 1 through 256
  put("Hash for division remainder = "); put(HA); new_line;

  A := "Badd";
  B := "est1";
  Sum := ConvertString4(A) + ConvertString4(B);
  put("The sum of ""Badd"" + ""est1"" is "); put(Sum); new_line;

  HA := abs(Sum) mod 256 + 1;
  put("Hash for division remainder = "); put(HA); new_line(2);

  -- Square and extract n bits for n = 8, table = 256.
  -- Note that on the PC a long integer is 32 bits. Assume we want
  -- the middle 8 bits. Must drop low order and high order 12 bits.
  put("Sample code for portion of square and extract N bits");
  new_line;
  HA := abs(Sum); -- Assume the square is the sum of "Badd" + "est1".
  put("The sum of 'Badd' + 'est1' = "); put(HA); new_line;
  HA := HA * 2**12; -- Get rid of 12 high order bits, shift left 12.
  put("Lose high order 12 bits HA * 2**12 = "); put(HA); new_line;

```

```

HA := (HA / 2**24); -- Move to low order 8 bits, shift right 24
put("HA = Middle 8 bits (HA / 2**24) = "); put(HA); new_line;
end;

```

```

$$$$$$$$$$$$$$$$$$$$ output *****
The sum of "ABCD" + "DEFG" is -1937078138
Hash for division remainder =    123
The sum of "Badd" + "est1" is -1780951897
Hash for division remainder =    90

```

```

Sample code for portion of square and extract N
bits
The sum of 'Badd' + 'est' = -1780951897
Lose high order 12 bits HA * 2**12 = 1924501504
HA = Middle 8 bits (HA / 2**24)=    114

```



```

--file UnCheck2.adb ** Compensate for runtime detecting integer overflow.
with Ada.Unchecked_Conversion, Ada.Text_IO; use Ada.Text_IO;
procedure UnCheck2 is
  package LongIntIO is new
Ada.Text_IO.Integer_IO(Standard.Long_Integer);
  use LongIntIO;

  type String4 is new String(1..4);
  function ConvertString4 is new
Ada.Unchecked_Conversion(String4, Long_Integer);

  A: String4 := "ABCD";
  B: String4 := "EFGH";
  Sum, HA: Long_Integer;
Begin
  -- "Folding" sample
  Sum := ConvertString4(A) / 3 + ConvertString4(B) / 3;
  put("The sum of ""ABCD"" + ""DEFG"" divided by 3 is ");
  put(Sum); new_line;

  -- Division remainder with folding and table size = 256.
  HA := abs(Sum) mod 256 + 1;
  put("Hash for division remainder = "); put(HA); new_line;

  A := "Badd";
  B := "est1";

  -- Fold: divide then add to avoid overflow.
  Sum := ConvertString4(A) / 2**13 + ConvertString4(B) / 2**13;
  -- dividing by 2**13 will likely cause collisions! Why?
  put("The sum of ""Badd"" + ""est1"" divided by 2**13 is ");
  put(Sum); new_line;

  HA := abs(Sum) mod 256 + 1;
  put("Hash for division remainder = "); put(HA); new_line(2);

```

```

-- Square and extract n bits for n = 8, table = 256.
-- Note that on the PC a long integer is 32 bits. Assume we want
-- the middle 8 bits. Must drop low order and high order 12 bits.
put("Sample code for portion of square and extract N bits");
new_line;
HA := abs(Sum); -- Assume the square is the sum of "Badd" + "est1".
put("The sum of ('Badd' + 'est1') / 2**13 = "); put(HA); new_line; -- 306886
HA := HA * 2**12; -- Get rid of 12 high order bits.
put("Lose high order 12 bits HA * 2**12 = "); put(HA); -- 1257005056
new_line;

HA := (HA / 2**24); -- Move to low order 8 bits. -- 74
put("HA = Middle 8 bits (HA / 2**24) = "); put(HA); new_line;
end;

```

```

$$$$$$$$$$$$$$$$$$$$ output *****
The sum of "ABCD" + "DEFG" divided by 3 is
785963052
Hash for division remainder =      45

```

```

The sum of "Badd" + "est1" divided by 2**13 is
306886
Hash for division remainder =      199

```

```

Sample code for portion of square and extract N bits
The sum of 'Badd' + 'est' / 2**13 = 306886
Lose high order 12 bits HA * 2**12 = 1257005056
HA = Middle 8 bits (HA / 2**24)=      74

```



# Collision Handling

Assume a collision has occurred. We must find another location in the table to place the key. In general, the probability of a collision is a function of how full the table is, not the size of the table. The fullness of a table is measured by its load factor defined as  $\alpha = (\text{number of entries in table}) / (\text{table size})$ . Note that  $\alpha$  is a fraction between 0 and 1. The larger the value of alpha, the fuller the table and hence the greater the probability of a collision when inserting a random key.

## Three Considerations Associated with Collisions!

- 1) For insertion, we must find another place to insert the record with a colliding key.
- 2) When we delete a record, can the space and possibly adjacent space be reclaimed?
- 3) If a record is deleted, is it possible to reduce the time to find records that might have passed through the location due to a deletion?

## Effect of Collisions:

Example 1:

Assume a table of size 256 containing 256 keys. Assume 200 keys have unique addresses, 56 keys collide once (require two probes). The the expected number of probes is  $E = (1 \cdot 200 + 2 \cdot 56) / 256 = 1.218$  probes.

Example 2:

Assume a table of size 256 containing 256 keys in sorted order. Using a binary search, the number of probes is  $\log_2 256 = 8$  maximum. The average is nearly the same as the maximum, or approximately 7.???

## Collision handling using Linear Probe

Assume a table of size TS with locations from 1 through TS and that the current key has hashed to location N where  $N \leq TS$ . Further assume that location N is currently occupied, a collision has occurred. To find a new location start searching linearly in locations N+1, N+2, etc. until a free location is found in the hash table. Place the new key in the first free location encountered. If the bottom of the table is reached without finding a free slot, wrap around to location 1 and start over. If the entire table is searched without finding a free location, then the table is full and the key may not be accepted. Placing keys in the next available location tends to fill localized portions of the table increasing the probability of a collision for keys hashing into this portion of the table. The result is called primary clustering due to excessive collisions as the table fills. The following formula and table demonstrate the expected performance of hashing with the linear probe technique as a function of the load factor.

**$E = (1 - \alpha / 2) / (1 - \alpha)$**  where  $\alpha = (\text{number keys in the table}) / (\text{table size})$ .

| Load Factor $\alpha$ | E (expected probes to locate key) |
|----------------------|-----------------------------------|
| 0.1                  | 1.06                              |
| 0.5                  | 1.50                              |
| 0.75                 | 2.5                               |
| 0.9                  | 5.50                              |

Consider the following insertions using the linear probe technique to handle collisions.

|   |      |       |       |       |       |       |
|---|------|-------|-------|-------|-------|-------|
| 1 | Sara | Sara  | Sara  | Sara  | &&&   |       |
| 2 |      |       |       | Mary  | Mary  |       |
| 3 |      |       |       |       |       |       |
| 4 | Tom  | Tom   | &&&   | Tom   | Tom   | Tom   |
| 5 |      | Ken   | Ken   | Ken   | Ken   | Ken   |
| 6 | Tim  | Tim   | Tim   | Tim   | Tim   | Tim   |
| 7 |      | Barby | Barby | Barby | Barby | Barby |
| 8 | Bob  | Bob   | Bob   | Bob   | Bob   | Bob   |
|   | (A)  | (B)   | (C)   | (D)   | (E)   | (F)   |

Assume that all entries in the table have been initialized to spaces to indicate that nothing has ever been stored in the table cells. Insert Tim at hash address 6, Bob at 8, Tom at 4 and Sara at 1. These operations are reflected in diagram A. Now insert Ken at location 4. This results in a collision and Ken must be moved to location 5. Note that in the future, it will require 2 probes to find Ken first at 4, then at 5. Find Hope at 2 and Tammy at 4. Now insert Barby at 7. The table now appears as in diagram B.

Delete Tom at location 4. Note that we can not just replace the value of Tom by spaces. If we do, we will no longer be able to locate Ken. If we look in location 4 and nothing is there, the appropriate conclusion is that Ken is not in the table. To solve this problem, we replace Tom by “&.” The & indicates that the space is available for future insertions but that it was previously occupied and another key may have passed through this position due to a collision. This is reflected by diagram C. If the cell below Tom at location 5 had been spaces, then nothing could have ever passed through location 4 due to a collision and we could have replaced Tom at location 4 by spaces.

Now insert Mary at hash address 7. Note that a collision occurs at locations 7, 8, and 1 (wrap around). Hence Mary must be placed in location 2 (diagram D).

Delete Sara. This is reflected in diagram E. Now delete Mary. Note that the space occupied by Mary and previously occupied by Sara can be set to spaces as no collisions could have passed through these locations (diagram F). It is important to do this to reduce the time spent in failed searches, i.e., looking for keys not in the table such as Zoro. Assume Zoro hashed to location 4. We would have to look in positions 4, 5, 6, 7, 8, and 1 before discovering Zoro is not in the table. Had we failed to set positions 1 and 2 to spaces after deleting Sara and Mary we would have had to look in positions 4, 5, 6, 7, 8, 1, 2, and 3 before discovering that Zoro is not in the table.

## Collision handling using Random Probe

Assume a table of size TS with locations from 1 through TS and that the current key has hashed to location N where  $N \leq TS$ . Further assume that location N is currently occupied, a collision has occurred. Generate a random offset C and probe the table at  $\rho_1 + HA$ . If the location is vacant, insert the key. If the location is filled, generate a random offset  $\rho_2$  and look in  $HA + \rho_2$ . Continue in this fashion till a vacant entry is found. In the future a collision will occur again when the key is originally hashed into the table. **Note that the same sequence of random numbers will be required during the search phase as was generated during the storage phase to find the key. Hence we desire a repeatable sequence of pseudo random numbers from 1 through the table size – 1 without repetition to be generated upon demand.** The table is full after looking at table size – 1 collisions. The performance of the random probe technique may be summarized as follows.

**$E = -(1/\alpha) \ln(1 - \alpha)$**  From "Scatter Storage Techniques" by Robert Morris, Bell Labs, CACM Digital Library, Vol. 11, Issue 1, 1968. ("ln" is the natural log).

| Load Factor $\alpha$ | E (expected probes to locate key) |
|----------------------|-----------------------------------|
| 0.1                  | 1.05                              |
| 0.5                  | 1.39                              |
| 0.75                 | 1.83                              |
| 0.9                  | 2.56                              |

- "ln" => natural log

By randomizing keys that collide across the table, search performance is improved. **The large primary clusters produced in the linear probe technique are avoided but smaller secondary clustering does occur.**

An algorithm for generating appropriate pseudo random numbers by Mauer follows that meets all qualifications.

# Random Numbers for use with Random Probe Collision Handling

Random number generator guaranteed to generate all numbers between one and the table size minus one without repetition. Based on an algorithm by Maurer, Vol. 11, No. 1, June 1965, CACM (1968 ACM Digital Archives). The table size must be a power of two. If the table size is not a power of two, discard all random numbers greater than or equal to the desired table size. The algorithm terminates when R is set to one by the calculations after the initialization. This implies that the random number generator has been through its entire range (period) from 1 to the table size – 1.

## Algorithm:

- 1) Set  $R \leftarrow 1$  (initialization - seed).
- 2) Every call set  $R \leftarrow R * 5$
- 3) Mask out all but low order (n+2) bits of product and place result in R.  $R \leftarrow \text{Mod}(5 * R, 2^{(n+2)})$
- 4) Set random number  $\rho \leftarrow R / 4$ .

Example: Assume the table size TS = 8, hence  $2^{(n)} = 8$  for  $n = 3$ .

|                                                                                                                                                  |                                                                                                                                   |
|--------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| $R \leftarrow 1$<br>$R \leftarrow 1 * 5 \leftarrow 5$<br>$R \leftarrow \text{Mod}(5, 32) \leftarrow 5$<br>$\rho_1 \leftarrow 5 / 4 \leftarrow 1$ | $R \leftarrow 17 * 5 \leftarrow 85$<br>$R \leftarrow \text{Mod}(85, 32) \leftarrow 21$<br>$\rho_5 \leftarrow 21 / 4 \leftarrow 5$ |
| $R \leftarrow 5 * 5 \leftarrow 25$<br>$R \leftarrow \text{Mod}(25, 32) \leftarrow 25$<br>$\rho_2 \leftarrow 25 / 4 \leftarrow 6$                 | $R \leftarrow 21 * 5 \leftarrow 105$<br>$R \leftarrow \text{Mod}(105, 32) \leftarrow 9$<br>$\rho_6 \leftarrow 9 / 4 \leftarrow 2$ |
| $R \leftarrow 25 * 5 \leftarrow 125$<br>$R \leftarrow \text{Mod}(125, 32) \leftarrow 29$<br>$\rho_3 \leftarrow 29 / 4 \leftarrow 7$              | $R \leftarrow 9 * 5 \leftarrow 45$<br>$R \leftarrow \text{Mod}(45, 32) \leftarrow 13$<br>$\rho_7 \leftarrow 13 / 4 \leftarrow 3$  |
| $R \leftarrow 29 * 5 \leftarrow 145$<br>$R \leftarrow \text{Mod}(145, 32) \leftarrow 17$<br>$\rho_4 \leftarrow 17 / 4 \leftarrow 4$              | $R \leftarrow 13 * 5 \leftarrow 65$<br>$R \leftarrow \text{Mod}(65, 32) \leftarrow 1$<br><b>The algorithm terminates.</b>         |

This algorithm generates a repeatable sequence of pseudo random numbers from 1 through the table size – 1 without repetition upon demand





```

declare
  HashKey: Integer := 5; --A random key to look for.
  HashAddress, HA: Integer;
  HashTable: array(1.. (2**TS)) of Integer := (others => 3); -- Initialize to full.
  KountPeeks: Integer;
  -- Create package to generate random offsets in the desired range.
  package RandomOffset is new URandInt(TS); use RandomOffset;
begin
  InitialRandInteger; -- Print range of random numbers for demonstration.
  for J in 1.. NLoops loop I := UniqueRandInteger; put(I); new_line; end loop;

  put("Print hash addresses generated starting with location 4."); new_line;

  HashAddress := 4; -- Random value for purpose of demonstration..

  -- Look in table at location HashAddress, if key matches report found;
  -- if the location is empty (0), insert the key; if the key does not match
  -- and the location is occupied, generate a random offset and look again
  -- table size minus one times (else the table is full). Note that the
  -- offset must always be added to the original hash address to prevent
  -- visiting the same location more than once.

  KountPeeks := 1; HA := HashAddress; InitialRandInteger;
  While (KountPeeks <= (2**TS )) loop
    -- Look at first address and (table size -1) offsets.
    put("Trip "); put(KountPeeks); put(" HA = "); put(HA); new_line;

    if HashTable(HA) = HashKey then
      null; --Insert code to process the key just found, i.e., put("The key exists!");
    elsif HashTable(HA) = 0 then
      null; --Insert code to insert the new key in the table, i.e., HashTable(HA) := HashKey.
      --or report the key is not in the table.
    else -- A collision has occurred. Generate a random offset and try again.
      KountPeeks := KountPeeks + 1;
      HA := HashAddress + UniqueRandInteger;
      if HA > (2**TS) then -- Wrap around HA exceeds size of the table.
        HA := HA - (2**TS);
      end if;
    end if;
  end loop;
end;

```

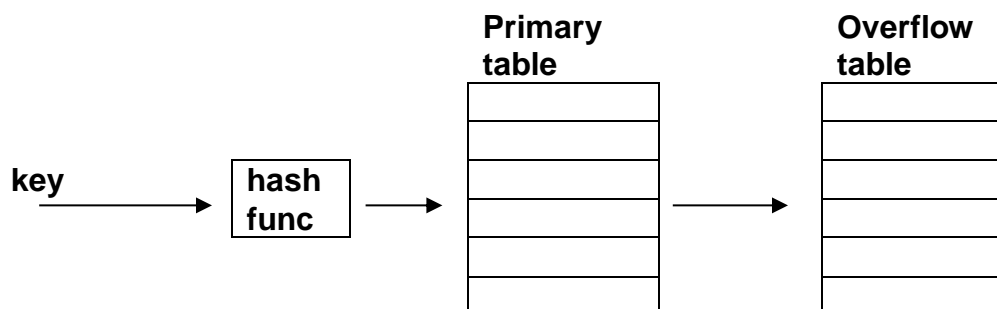


There are several variations of direct chaining. One must be careful that all concerned individuals are actually talking about the same method. One method is to treat the hash table as a set of list heads for a linked list into an auxiliary storage area. Each time a key is hashed, it is chained (linked list) in the auxiliary storage area to all other records having the same key. A second method referred to as direct chaining is implemented in the following manner. Each entry in the hash table contains not only the key and record but a link field initially set to null. When a collision occurs during storage, an address to place the key is found using any desired technique. Once the record is stored, it is placed on a linked list of all records colliding at the original hash address. In the future during lookup, only records on the linked list need be searched. This should reduce the search time as the table gets full as it is not necessary to examine records that can not satisfy the search criteria.

### Collision handling using Direct Chaining with an auxiliary overflow area to allow the number of keys to exceed the size of the primary hash table.

$$E = 1 + \alpha / 2$$

| Load Factor $\alpha$ | E (expected probes to locate key) |
|----------------------|-----------------------------------|
| 0.1                  | 1.05                              |
| 0.5                  | 1.25                              |
| 0.75                 | 1.38                              |
| 0.9                  | 1.45                              |
| 1.5 => 150% full     | 1.75                              |
| 2.0 => 200% full     | 2.00                              |



SNOBOL language interpreter symbol table consists of 50 entries. If you have 100 variables and a reasonable hash function then there will be about 2 entries in the overflow table for each hash address. First hash the key then search the linked list. Note this will require 1 to 2 probes in the list for any random variable name! Interpreters must refer to the symbol table on each variable reference.

# Java Hash Tables

**Note the probability of a new key causing a collision is dependent on how full the table is, not how many keys are in the table.**

**The default table size is 101.** As the table gets fuller the probability of a collision increases. The fullness of the table is usually expressed as the load factor  $\alpha = (\text{number in table}) / (\text{table size})$ . **The default load factor is 75%.** When the number of occupied slots in the hash table becomes more than the capacity times the load factor, the size of the table is automatically doubled and the keys re-hashed.

Java recommends that the table size be a prime number or at least an odd number (division remainder technique).

In general, the hash code generated by Java for integers and string is based on the value of the integer or string. Java uses the memory location of objects to compute a hash address.

For example:

```
String s = "Ok";  
System.out.println(s.hashCode()); // prints 3030.
```

Samples:

| String | Hash code | Table Address * |
|--------|-----------|-----------------|
| Ok     | 3030      | $0 + 1 = 1$     |
| Hello  | 140207504 | $11 + 1 = 12$   |
| Harry  | 140013338 | $61 + 1 = 62$   |
| Hacker | 88475206  | $23 + 1 = 24$   |

\* Assumes a table of size 101 with locations from 1 through 101.

In the division remainder technique, divide the key by the table size producing a quotient and remainder. Discard the quotient and use the remainder as the hash address. Note:  $0 \leq \text{hash address} < \text{table size}$ . For a hash table with locations 1 through the table size, you must add one to the remainder to obtain the actual hash address. Hence a hash address  $HA = \text{remainder} + 1$ .

$3030 / 101$  gives  $q = 30$  with  $r = 0$ . Implies  $HA = 0 + 1 = 1$ .

$140207504 / 101$  gives  $q = 1388193$  with  $r = 11$ .  $HA = 11 + 1 = 12$ .

```
// Demonstrates class Hashtable of the java.util package.
```

```
import java.util.*;  
import java.awt.*;  
import java.awt.event.*;  
import java.applet.Applet;
```

```
public class Hashtable1 extends Applet
```

```
    implements ActionListener {  
    private Hashtable table;  
    private TextField surName, givenName, ageField;  
    private TextArea display;  
    private Button put, get, listElems;
```

```
public void init () {
```

```
    table = new Hashtable( ); // Default size = 101, load factor = 75%.  
    // When the occupied slots becomes greater than the capacity * the load  
    factor,
```

```
    // the table is re-allocated with the size doubled and all keys are re-hashed.
```

```
    // Hashtable table2 = new Hashtable(table size, load factor). The table size
```

```
    // should be a prime number or at least an odd number not evenly
```

```
    // divisible by small integers.
```

```
    add( new Label( "First name" ) ); givenName = new TextField( 8 ); add( givenName  
);
```

```
    add( new Label( "    Last name (key)" ) ); surName = new TextField( 8 ); add(  
surName );
```

```
    add( new Label( " Age" ) ); ageField = new TextField( 4 ); add( ageField);
```

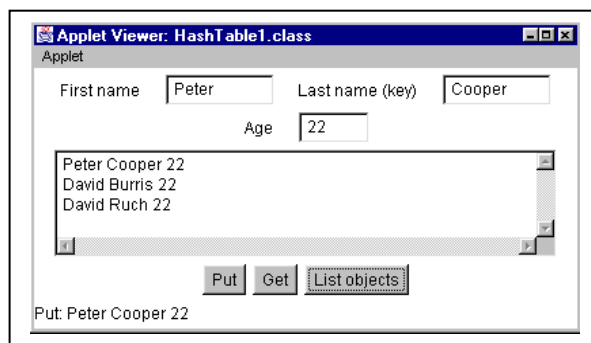
```
    display = new TextArea( 4, 50 ); add( display );
```

```
    put = new Button( "Put" ); put.addActionListener( this ); add( put );
```

```
    get = new Button( "Get" ); get.addActionListener( this ); add( get );
```

```
    listElems = new Button( "List objects" ); listElems.addActionListener( this );  
    add( listElems );
```

```
}
```



```

public void actionPerformed((ActionEvent e) {
    if ( e.getSource() == put ) { // Create and add an object to the hash
table.
        Employee emp = new Employee( givenName.getText(),
                                    surName.getText(),
                                    Integer.parseInt(ageField.getText()) );

        Object val = table.put( surName.getText(), emp );
        // Actual add using key and object.

        if ( val == null ) // "put" returns null if the table at the hash address was empty.
            showStatus( "Put: " + emp.toString() );
        else
            showStatus( "Put: " + emp.toString() + "; Replaced: " + val.toString() );
    }
    else if ( e.getSource() == get ) { // Retrieve entry specified by the current "key."
        Object val = table.get( surName.getText() );

        if ( val != null ) // "get" returns null if the key is not in the table.
            showStatus( "Get: " + val.toString() );
        else
            showStatus( "Get: " + surName.getText() + " not in table" );
    }
    else if ( e.getSource() == listElems ) { // List all objects in the hash table.
        StringBuffer buf = new StringBuffer( );

        for (Enumeration enum = table.elements();
             enum.hasMoreElements(); )
            buf.append( enum.nextElement() ).append( '\n' );

        display.setText( buf.toString() );
    }
}
}

```

```

class Employee {
    private String first;
    private String last;
    private int age;

    public Employee( String givenName, String surName,
                    int ageIn ) {
        first = givenName;
        last = surName;
        age = ageIn;
    }

    public String getGivenName( ) { return first; } // Object Access functions.
    public String getSurName( ) { return last; }
    public int getAge( ) { return age; }

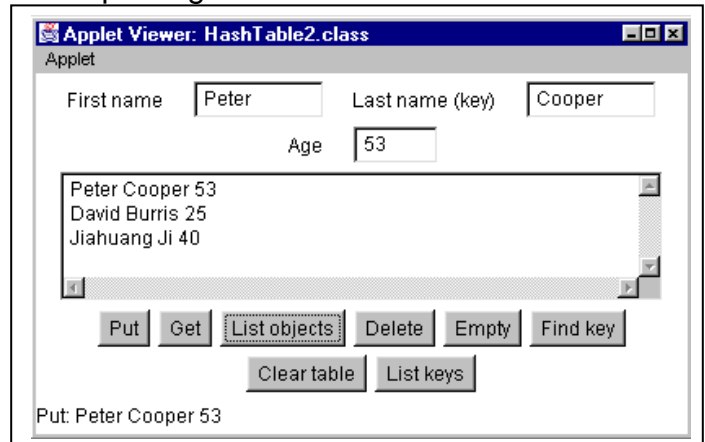
    public String toString( ) { return first + " " + last + " " + age; }
}

```

// Demonstrates class Hashtable of the java.util package.

```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;
```

```
public class Hashtable2 extends Applet
```



```
    implements ActionListener {
    private Hashtable table;
    private TextField surName, givenName, ageField;
    private TextArea display;
    private Button put, get, listElems, deleteKey, empty, findKey, clearTable, clearKey,
    listKeys;

    public void init() {
        table = new Hashtable(); // Default size = 101, load factor = 75%.
        // When the occupied slots becomes greater than the capacity * the load
        factor,
        // the table is re-allocated with the size doubled and all keys are re-hashed.
        // Hashtable table2 = new Hashtable(table size, load factor). The table size
        // should be a prime number or at least an odd number not evenly divisible by
        // by small integers.

        add( new Label( "First name" ) ); givenName = new TextField( 8 ); add( givenName
    );
        add( new Label( "   Last name (key)" ) ); surName = new TextField( 8 ); add(
    surName );
        add( new Label( " Age" ) ); ageField = new TextField( 4 ); add( ageField);
        display = new TextArea( 4, 50 ); add( display );
        put = new Button( "Put" ); put.addActionListener( this ); add( put );
        get = new Button( "Get" ); get.addActionListener( this ); add( get );
        listElems = new Button( "List objects" ); listElems.addActionListener( this );
        add( listElems );
        deleteKey = new Button( "Delete" ); deleteKey.addActionListener( this ); add(
    deleteKey );
        empty = new Button( "Empty" ); empty.addActionListener( this ); add( empty
    );
        findKey = new Button( "Find key" ); findKey.addActionListener( this ); add( findKey
    );
        clearTable = new Button( "Clear table" ); clearTable.addActionListener( this
```

```

);
add( clearTable );
listKeys = new Button( "List keys" ); listKeys.addActionListener( this ); add( listKeys
);
}

public void actionPerformed((ActionEvent e) ) {
    if ( e.getSource() == put ) { // Create and add an object to the hash table.
        Employee emp = new Employee( givenName.getText(),
                                     surName.getText(),
                                     Integer.parseInt(ageField.getText()) );

        Object val = table.put( surName.getText(), emp ); // Actual add using key and
object.

        if ( val == null ) // "put" returns null if the table at the hash address was empty.
            showStatus( "Put: " + emp.toString() );
        else
            showStatus( "Put: " + emp.toString() +
                        "; Replaced: " + val.toString() );
    }
    else if ( e.getSource() == get ) { // Retrieve the entry specified by the current "key."
        Object val = table.get( surName.getText() );

        if ( val != null ) // "get" returns null if the key is not in the table.
            showStatus( "Get: " + val.toString() );
        else
            showStatus( "Get: " + surName.getText() + " not in table" );
    }
    else if ( e.getSource() == listElems ) { // List all objects in the hash table.
        StringBuffer buf = new StringBuffer();

        for ( Enumeration enum = table.elements(); enum.hasMoreElements(); )
            buf.append( enum.nextElement() ).append( '\n' );

        display.setText( buf.toString() );
    }

    else if ( e.getSource() == deleteKey ) { // Delete object with current key
        Object val = table.remove( surName.getText() );

        if ( val != null ) // Object pointer is null if successfully deleted.
            showStatus( "Deleted: " + val.toString() );
        else
            showStatus( "Key: " + surName.getText() + " not in table, delete failed." );
    }
    else if ( e.getSource() == empty ) {

```

```

        showStatus( "Empty: " + table.isEmpty() );
    }
    else if ( e.getSource() == findKey ) { // Is the key in the table?
        showStatus( "Key: " + table.containsKey( surName.getText() ) );
    }
    else if ( e.getSource() == clearTable ) { // Erase all entries from the table.
        table.clear(); showStatus( "Table cleared: Table is now empty" );
    }
    else if ( e.getSource() == listElems ) { // Enumerate all objects in the table.
        StringBuffer buf = new StringBuffer();

        for ( Enumeration enum = table.elements(); enum.hasMoreElements(); )
            buf.append( enum.nextElement() ).append( '\n' );
        display.setText( buf.toString() );
    }
    else if ( e.getSource() == listKeys ) { // List all keys in the table.
        StringBuffer buf = new StringBuffer();

        for ( Enumeration enum = table.keys(); enum.hasMoreElements(); )
            buf.append( enum.nextElement() ).append( '\n' );
        display.setText( buf.toString() );
    }
}
}
}

```

```

class Employee {
    private String first;
    private String last;
    private int age;

    public Employee( String givenName, String surName, int ageIn )
    {
        first = givenName;
        last = surName;
        age = ageIn;
    }

    public String getGivenName() { return first; } // Object Access functions.
    public String getSurName() { return last; }
    public int getAge() { return age; }

    public String toString() { return first + " " + last + " " + age; }
}

```



# **File Organization**

## **And**

# **Access Methods**

### **Organization:**

- 1) **Sequential**
- 2) **Indexed**
- 3) **Relative**
- 4) **Text Files**

### **Access Methods:**

- 1) **Sequential**
- 2) **Random**
- 3) **Dynamic**

--file: RelativeFiles.doc

# **Random Access Methods**

Records are stored and retrieved on the basis of a predictable relationship between the key of the desired record and the location where the record is stored.

## **Methods**

- 1) **Direct:** The user supplies either the direct address on the storage device of the desired record (e.g., on a disk the cylinder, track, and sector) or a virtual address that can be transformed easily to the actual address either by the system software or device hardware (e.g., a relative record number for conversion).
- 2) **Dictionary Look-up.**
- 3) **Hashing or Calculation.**

# Relative Files

(fixed or variable length)

## Traditional – Fixed Length Records

All space for file is typically allocated as a contiguous unit at the time the file is created.

|                |       |      |     |     |     |     |     |
|----------------|-------|------|-----|-----|-----|-----|-----|
|                | Betty | Adam |     | Sue |     | Tom | ... |
| Record Number  | 0     | 1    | 2   | 3   | 4   | 5   |     |
| Device Address | 0     | 50   | 100 | 150 | 200 | 250 |     |

$$\text{Loc}[\text{REC}_J] = \text{Base} + \text{Offset}$$

$$\text{Offset} = J * \text{number of characters in record}$$

**Example:** Find relative record 3 assuming 50 characters per record starting at a base address of zero on the storage device.

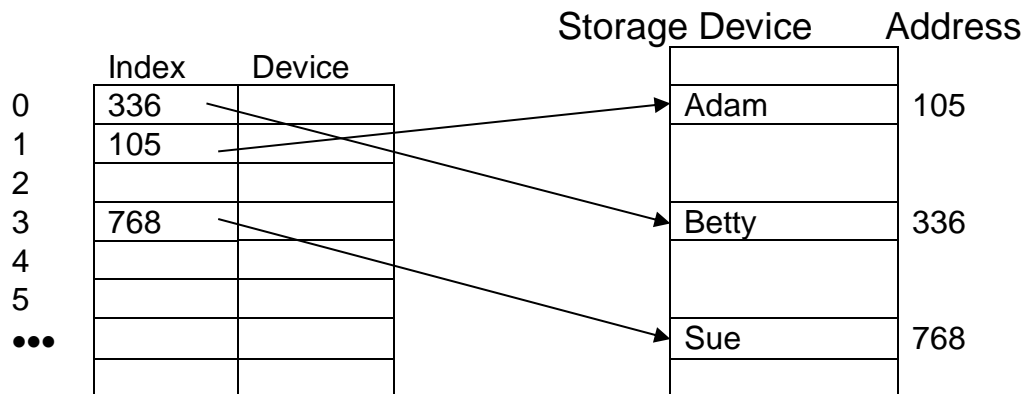
$$\text{LOC}[\text{REC}_3] = 0 + 3 * 50 = 150.$$

~~~~~

## Relative files may be based on a table lookup (B-Tree).

This is particularly attractive for:

- 1) variable length records,
- 2) in situations where dynamic storage allocation is desirable.



in Data Structures Programs page 94.

-- Demonstrate sequential I/O of records in Ada versus "stream I/O"  
-- (record oriented[business/scientific world], not text files).

with Ada.Text\_IO;  
**with sequential\_io;**  
procedure seqio is

type rec is record  
    i:integer;  
    a:string(1..5);  
end record;

**package hope is new sequential\_io(rec);**  
**use hope;**

**f1: file\_type;** -- or f1: hope.file\_type;  
**rec1:** rec;

begin

**hope.create(f1,out\_file,"joe");**  
**rec1.a := "joeis";**  
**rec1.i := 99;**  
**hope.write(f1,rec1);**  
**rec1.a := "samis";**  
**rec1.i := 23;**  
**hope.write(f1,rec1);**  
**hope.close(f1);**

**hope.open(f1,in\_file,"joe");**  
while not **end\_of\_file(f1)** loop  
    **hope.read(f1,rec1);**  
    **-- process record**  
end loop;

Ada.Text\_IO.put("hi there");  
**hope.close(f1);**

end;

```

with direct_io; //in file mkdirect.ada & directio.doc
with text_io;
procedure mkdirect is
    type rec is record
        i:integer;
        a:string(1..5);
    end record;

    package iio is new text_io.integer_io(integer);
    use iio;

    package io_direct is new direct_io(rec);
    use io_direct;

    pt: positive_count;      -- 1, 2, 3, ...
    f1:file_type;

    rec1: rec;
    j: integer := 0;

begin
    create(f1, inout_file, "joedir");
    rec1.a := "abcde";
    reset(f1);
    -- COUNT and POSITIVE_COUNT are defined in package direct_io;
    for pt in positive_count range 1..10 loop
        j := j + 1;
        rec1.i := j;
        write(f1, rec1, pt);
    end loop;
    close(f1);

    open(f1,inout_file,"joedir");
    reset(f1);
    for pt in positive_count range 1..10 loop
        read(f1, rec1, pt);
        text_io.put(rec1.a); iio.put(rec1.i); text_io.new_line;
    end loop;
    close(f1);
end;

```

## Queuing Disk Access Request

Assume a moving arm disk on a multi-user system. The time to read (write) data requires moving the read/written head to the proper track (seek time) plus the rotational delay to rotate under the read/write head (about  $\frac{1}{2}$  revolution) plus the transport time to read (or write) the data passing under the read/write head. It is not unusual for 90% or more of the time to be seek time. The total time is frequently expressed as  $\text{time} = T_{\text{seek}} + T_{\text{rotational-delay}} + T_{\text{transport}}$ . Disk access times reported by manufacturers are typically weighted averages.

User request to read/write the disk are placed in a traditional queue (FIFO) for service by the operating system. Assume it requires 0.1 millisecond to read or write information if the disk is on the right track. Further assume that it takes 1 millisecond per track to move the read / write head from one track to the next (including the time for the actual read or write. If the read / write head starts at track 1, the time to process the following request in the order they arrived for the indicated tracks is:

Track request in order of arrival (democratic): 1, 20, 5, 12, 12, and 13.

Processing time =  $0.1 + (20-1)*1 + (20-5)*1 + (12-5)*1 + 0.1 + (13-12)*1 = 42.2$  milliseconds.

Now assume that while the current disk request is being processed, new read/write request are placed in a priority queue as they arrive. Highest priority will be given to the next disk request that requires the smallest disk arm movement in an attempt to reduce the total time required to process all request. Hence the above queue would be ordered as follows:

Priority Queue: 1, 5, 12, 12, 13, and 20.

Processing time =  $0.1 + (5-1)*1 + (12-5)*1 + 0.1 + (13-12)*1 + (20-13)*1 = 19.2$  milliseconds. While not democratic, total processing time has been reduced by  $19.2/42.2*100\% = 49\%$ . The process of serving request to minimize disk arm movement and hence total processing time is known as “staging the disk.”

The question arises as to how to physically implement the priority queue. It could be maintained as a sequentially allocated list with new items inserted in priority order as opposed to simple insertion at the rear. If the queue is large, a binary search might be used to locate where the new arrival should

be placed. Unfortunately, if desired tracks are random, approximately half the list would have to be moved each time to make space for the new arrival. A linked list would reduce the insertion overhead required to move existing entries but we would lose the advantage afforded by the binary search. An interesting alternative using sequentially allocated storage for the priority queue with insertion using a heap has many advantages.

**Priority Queue:** A special case of a Binary Trees represented as a Heap. A heap is a data structure where the parent is always larger than its children.

**Definitions:**

- 1) A binary tree  $T$  of depth  $K$  is “**complete**” if and only if each vertex at level  $K$  is a leaf and each vertex whose level is less than  $K$  has nonempty left and right children. Hence a complete binary tree has all its leaves at the same level and every non-leaf vertex has both children present. A complete binary tree of depth  $K$  always has exactly  $2^{(K+1)} - 1$  vertices. A tree consisting of one vertex (node) has depth 0; a complete binary tree of depth 1 has three vertices; a tree of depth 2 has seven vertices; and so on. Alternately, a complete binary tree with  $N$  vertices has a depth equal to  $\log_2 (N+1) - 1$ .

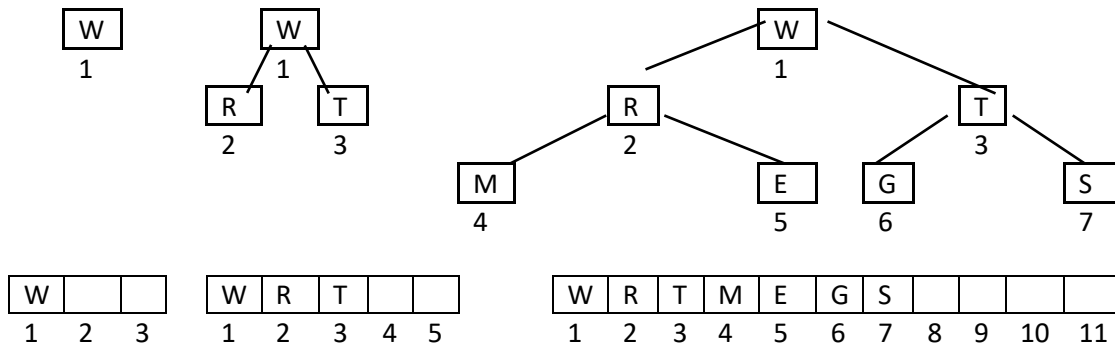
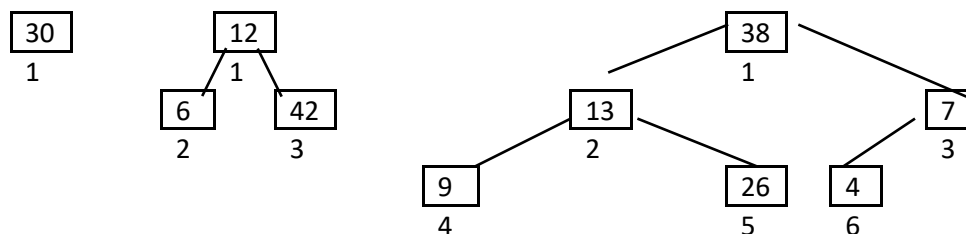


Figure 1. Three “complete” binary trees of depth 0, 1, and 2 with addresses for a linear array representation. Empty cells in the linear representation indicate entries not in use.

- 2) A binary tree  $T$  is an **almost-complete binary tree** (ACBT) of depth  $K$  if and only if it is either complete, or fails to be complete only because some of its leaves are at the right-hand end of level  $K - 1$ . This has the effect of concentrating all the leaves at level  $K$  to the left end of the level and all the level  $(K - 1)$  leaves at the right end of the tree. Note that a “complete” tree is also an ACBT.



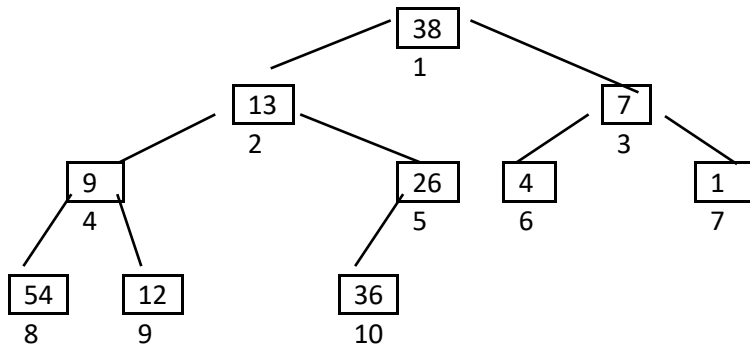
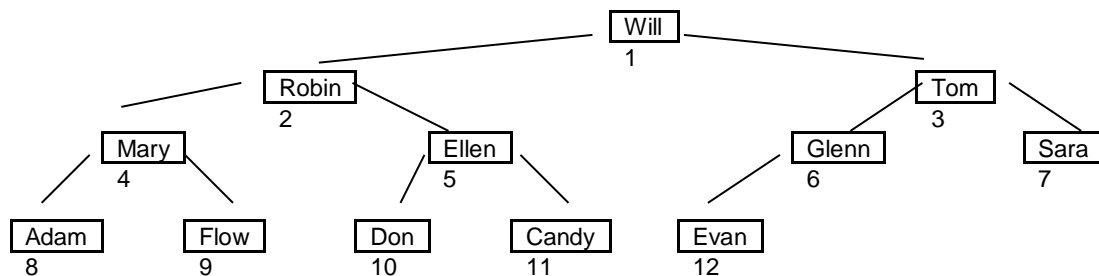


Figure 2. Several ACBT (almost complete binary trees, note they may be complete).

Traditionally binary trees are implemented using linked list. ACBT's have the useful property that they may be implemented as an array. Indeed, the first element of the array corresponds to the root of the tree; the second and third elements of the array correspond to the children of the root: and so on. Hence the children of vertex (node) at location  $J$  appear in the array at addresses  $(2 * J)$  and  $(2 * J) + 1$  for the left and right child respectively. Note the parent of any child is at address  $(J / 2)$  using integer division. Hence we can locate existing parents and children of any random vertex in the tree via a simple calculation.

For example, consider the following linear representation of the ACBT currently containing 12 elements compared to the logical tree below it.

|      |       |     |      |       |      |      |      |      |     |       |      |    |    |
|------|-------|-----|------|-------|------|------|------|------|-----|-------|------|----|----|
| Will | Robin | Tom | Mary | Ellen | Glen | Sara | Adam | Flow | Dan | Candy | Evan |    |    |
| 1    | 2     | 3   | 4    | 5     | 6    | 7    | 8    | 9    | 10  | 11    | 12   | 13 | 14 |



- 3) A “*heap*” is an ACBT whose key at every node is greater than or equal to the keys of its children. The above tree (both the linear representations and the logical representation (tree)) is a heap. Note that we do not require a left child to be smaller than a right child.

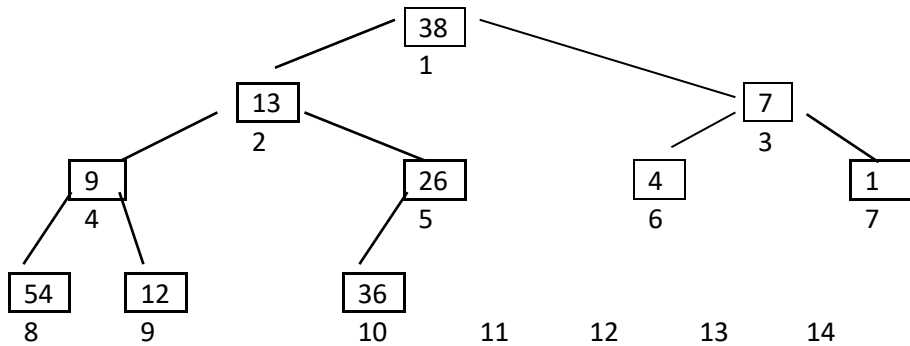
Heaps are generally built by adding one node at a time in the next available leaf position. In a linear representation containing  $M$  vertices, the next available leaf location is  $(M + 1)$ . Since a heap is always an ACBT, in a linear representation of the heap this position is always known. To maintain the heap property, the key of the new vertex is compared with its parent. If the key is larger than the parent, we must exchange the new vertex with its parent. This process is repeated until the new vertex is not larger than its parent. Note that 6 may be added to the following tree without having to move any vertices (nodes).



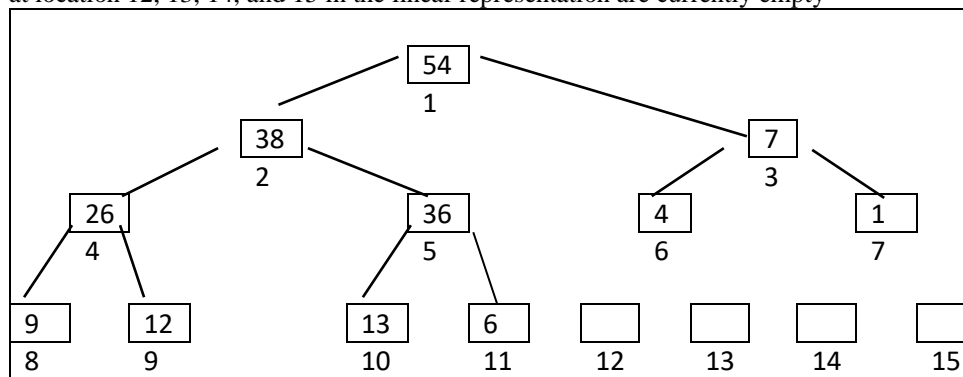
Assume data entered sequentially prior to creating the priority queue maintained as a heap in order show below. All data is physically in a sequentially allocated array.

|    |    |   |   |    |   |   |    |    |    |    |    |    |    |
|----|----|---|---|----|---|---|----|----|----|----|----|----|----|
| 38 | 13 | 7 | 9 | 26 | 4 | 1 | 54 | 12 | 36 |    |    |    |    |
| 1  | 2  | 3 | 4 | 5  | 6 | 7 | 8  | 9  | 10 | 11 | 12 | 13 | 14 |

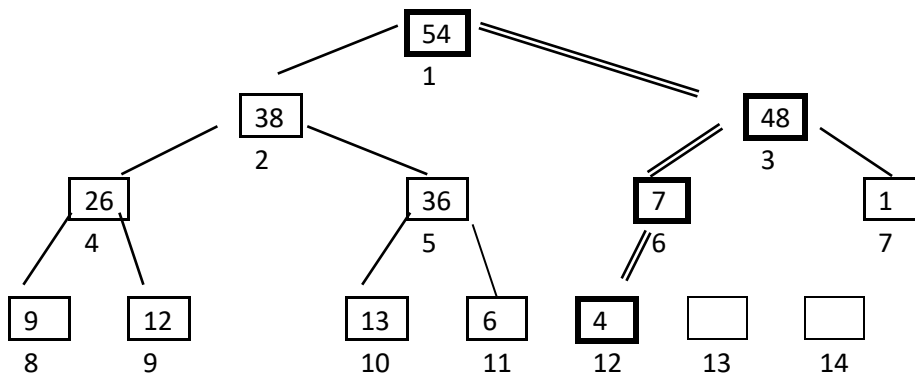
Prior to heap creation (Create using 38, 13, 7, 9, 26, 4, 1, 54, 12, and 36 with M = 10):



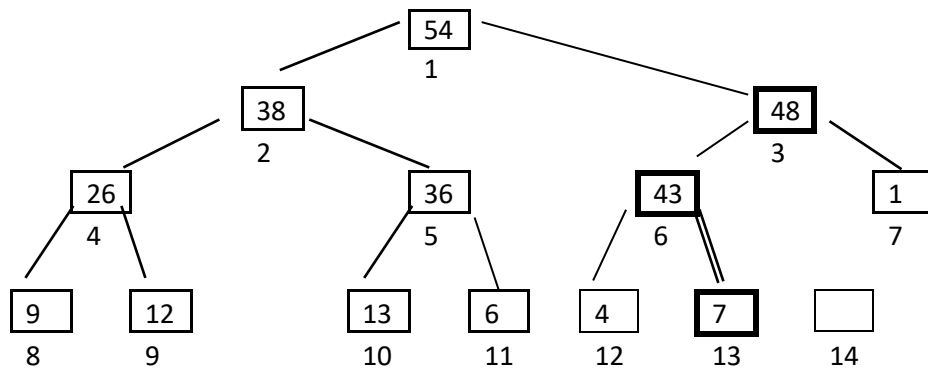
After creating the heap, inserting the key of 6 results in no changes. M = 11 after the insertion. The cells at location 12, 13, 14, and 15 in the linear representation are currently empty



Inserting the key 48 results in several exchanges (first 48 & 4, then 48 & 7). M = 12.



Finally, we insert a 43. Exchanges are 43 & 7.  $M = 13$ .



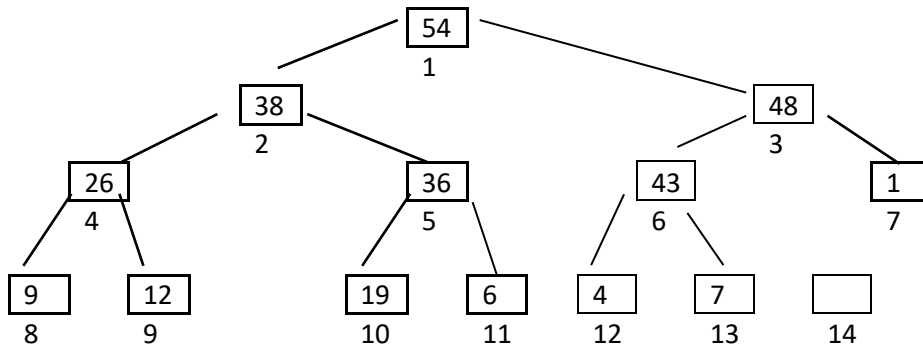
When we insert a new node into the heap the minimum number of node exchanges is 0. At most we must make approximately  $\log_2 N$  exchanges, hence we say the time for insertion is on the order of  $O(\log_2 N)$ .

- 4) A “**priority queue**” is a data structure that orders its elements (enqueued and dequeued) based on the value of some key, essentially a priority scheme. For example, in an operating system when new task are submitted to the system they are assigned a priority. This priority stays with the task (some systems modify priority dynamically to improve overall system performance or to assure that urgent tasks are accomplished expediently) as it traverses the system, e.g., completes an I/O operation and mover from the “wait state” to the “ready list.” When the system process scheduler selects the next task for execution, it commonly selects the task from the ready list with the highest priority. Disk reads and writes are queued in a similar manner. The problem is how to minimize the time spent in placing task on the ready list and finding the highest priority task when required. One approach would be to modify a traditional queue. When new task are place on the list, we place them in order by priority, descending key order. If the queue is kept as a sequentially allocated list, we must first search the list for the proper position sequentially then move all items whose keys have lesser values to make space for the higher priority key. The dequeue operation will be fast, simply remove the first item on the list since it is kept in descending order. The time to search sequentially however and to move nodes is expensive. The priority queue could be kept as a linked list. We would still have to search sequentially but the insertion would be more efficient. In either case the time to insert a new node will range from 1 probe to N probes, or time of order  $O(N)$ . This time includes both the search and insertion operations.

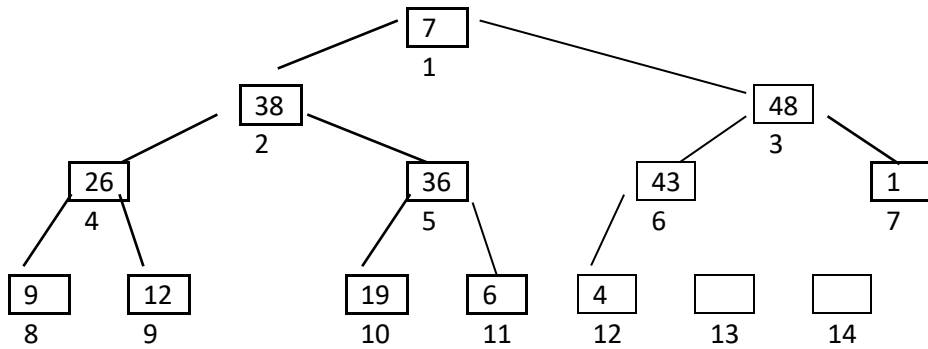
In many cases a priority queue may be maintained with far less overhead by using a heap. The top of the heap will be the largest key, location 1 in a linear representation. New entries in a priority queue of length M are always placed at location (M + 1). It then takes of  $O(\log_2 M)$  operations to move the new key up the heap to its final position. Hence a priority queue implemented as a heap is generally faster than other representations.

In a dequeue operation, after the largest element in the heap is removed (location 1 of the array representation), the heap element at location M is moved to location 1 and the size of the heap reduced by 1, i.e.,  $M = M - 1$ . The resulting data structure may not be a heap as the first element may no longer be larger than its children. It must be moved down the structure until it satisfies the conditions for a heap. This should require a maximum of  $O(\log_2 M)$  operations to complete. For example, assume the following heap represents process identification number for a Linux system. When the current process completes, exceeds its time slice, or blocks (e.g., I/O request), the system process scheduler selects the highest priority process as shown and reorganizes the heap. For convenience, we assume that the relative magnitude of the process identification number represents the priority of the process relative to other process in the system.

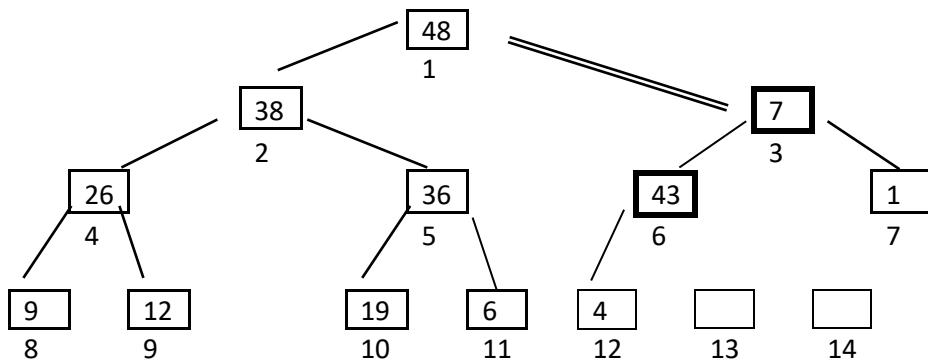
Process 54 is selected.



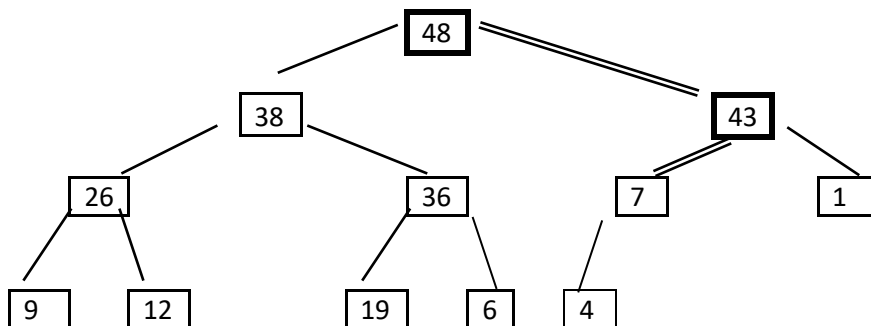
To recreate the heap move the key from position M to position 1, then  $M = M - 1 = 12$ .



Exchange 7 and 48



Exchange the 7 & 43. The Heap has been recreated.



Be using a “heap,” the overhead for insertion and deletion from the priority queue are on the order of  $\log_2 N$ .

The following programs implement the priority queue using Ada. The example is far more complex than necessary to implement a simple priority queue. The opportunity has been taken to maximize the flexibility of Abstract Data Types (ADT's) by utilizing generics. The queue element type is a generic parameter as is the subscript a person might use to access elements in the linear representation. Normally you would probably only considering using integers for subscripts which would greatly simplify the code. To implement this program in C++, the programmer would use templates for the Ada generics and use “function pointers” to pass functions to the templates (replace the “with function” in Ada). The C++ “enum” could be used to allow for subscripting with a programmer defined enumeration type as was done in the Ada code.

Suggest Sources for further reading (all used in creating this example):

- 1) “Searching and Sorting,” volume 3 of Knuth’s famous series, Addison-Wesley.
- 2) “Programming in Ada 95” by J.G. P. Barnes, Addison-Wesley.
- 3) “Software Construction and Data Structures with Ada 95” by Michael B. Feldman, Addison-Wesley.

In “UseGenericPriorityQueue” the value pairs (Mary, Jan); (Adam, Mar); (Evan, Dec); (Robin, Oct); and (Flow, Oct) are entered in the indicated order. We assume the name fields indicate priority based on ascending lexicographic order. The output from the priority queue is (Robin, Oct); (Mary, Jan); (Flow, Oct); (Evan, Dec); and (Adam, Mar). This demonstrates that a priority queue may also be used for sorting by inserting all records prior to removing them sequentially.

Compile in the following order:

- 1) GenericSwap.ads, GenericSwap.adb
- 2) GenericHeap.ads, GenericHeap.adb
- 3) GenericPriorityQueue.ads, GenericPriorityQueue.adb
- 4) UseGenericPriorityQueue.adb (This is the actual main program.)

#### **--Contents of GenericSwap.ads**

```
-----
-- Pre-condition: P1 and P2 are any object type except limited private
-- Post-condition: The values of P1 and P2 are swapped.
-----
```

GENERIC

```
    TYPE ValueType IS PRIVATE;  -- any type OK except LIMITED PRIVATE
```

```
PROCEDURE GenericSwap(P1, P2: IN OUT ValueType);
```

#### **--Contents of GenericSwap.adb**

```
procedure GenericSwap(P1, P2: IN OUT ValueType) IS
    Temp: ValueType;
begin -- Swap_Generic
    Temp := P1;
    P1    := P2;
    P2    := Temp;
end GenericSwap;
```

#### **Contents of GenericHeap.ads**

GENERIC

```
    TYPE KeyType IS PRIVATE;
    TYPE ElementType IS PRIVATE;
    TYPE IndexType IS RANGE <>;  -- integer subscripts
    TYPE ListType IS ARRAY (IndexType RANGE <>) OF ElementType;
    WITH FUNCTION KeyOf (Element: ElementType) RETURN KeyType IS <>;
    WITH FUNCTION "<"(Left, Right: KeyType) RETURN Boolean IS <>;
    with Procedure MyPut(x: KeyType);
```

PACKAGE GenericHeap IS

```
-----
-
--| Specification for Generic Heaps Package
--|
--| Last Modified: September 13, 2002
-----
-
```

```
PROCEDURE ExtendHeap(aHeap: IN OUT ListType);
-- Pre:  aHeap(aHeap'First..aHeap'Last-1) is a heap such that
--        aHeap(aHeap'First) is the "largest" element.
-- Post: extends heap by adding aHeap(aHeap'Last) to it.
```

```
PROCEDURE AlmostHeapToHeap(aHeap: IN OUT ListType);
-- Pre:  aHeap(aHeap'First..aHeap'Last) is an "almost heap",
--        that is, it would be a heap except that aHeap(aHeap'First) may
be
--        "smaller" than one or both of its children
```

```

-- Post: aHeap(aHeap'First..aHeap'Last) is a heap

END GenericHeap;

--Contents of GenericHeap.adb
WITH GenericSwap;
PACKAGE BODY GenericHeap IS
-----
-
--| Body of Generic Heaps Package
--| Author:
--| Last Modified: September 13, 2002
--| An "AlmostHeap" is a heap with only the top node out of place.
-----
-
  procedure Exchange is new GenericSwap(ValueType => ElementType);

  function ">=" (Left, Right: KeyType) return Boolean is
  begin
    return NOT (Left < Right);
  end ">=";

  procedure ExtendHeap(aHeap: IN OUT ListType) is

    Top      : constant Integer := Integer(aHeap'First);
    Child    : Integer;
    Parent   : Integer;
    IChild   : IndexType; -- must satisfy Ada type compatibility rules
    IParent  : IndexType;

  begin -- ExtendHeap body
    if aHeap'First = aHeap'Last then -- heap has single element
      return;
    end if;

    Child := Integer(aHeap'Last);
    Parent := Child / 2;

    while (Parent >= Top) loop
      IParent := IndexType(Parent);
      IChild := IndexType(Child);
      -- MyPut( KeyOf(aHeap(IParent)) ); --** use of generic I/O
routine.
      exit when (KeyOf(aHeap(IParent)) >= KeyOf(aHeap(IChild)) );
      Exchange(aHeap(IChild), aHeap(IParent));
      Child := Parent;
      Parent := Parent / 2;
    end loop;
  end ExtendHeap;

  procedure AlmostHeapToHeap(aHeap: IN OUT ListType) is

    Bottom : constant Integer := Integer(aHeap'Last);
    Parent : Integer;
    Child  : Integer;
    IParent: IndexType; -- for type Ada compatibility

```

```

    IChild : IndexType;
    Placed : Boolean := False;

begin -- AlmostHeapToHeap body

    if aHeap'First = aHeap'Last then -- only one element in heap
        return;
    end if;

    Parent := Integer(aHeap'First);
    Child  := Integer(aHeap'First) + 1;

    while (Child <= Bottom) AND NOT Placed loop

        IChild := IndexType(Child);
        IParent := IndexType(Parent);

        if Child+1 <= Bottom then      -- Parent has 2 Children

            IF      KeyOf(aHeap(IParent)) >= KeyOf(aHeap(IChild))
                AND KeyOf(aHeap(IParent)) >= KeyOf(aHeap(IChild + 1)) THEN
                Placed := True;
            ELSIF KeyOf(aHeap(IChild+1)) < KeyOf(aHeap(IndexType(Child)))
THEN
                Exchange(aHeap(IParent), aHeap(IChild));
                Parent := Child;  --left Child is larger than parent
                Child := 2 * Parent;
            ELSE
                Exchange(aHeap(IParent), aHeap(IChild+1));
                Parent := Child+1; --right Child is larger than parent
                Child := 2 * Parent;
            END IF;

            ELSE      -- Parent has only one Child
                IF KeyOf(aHeap(IParent)) < KeyOf(aHeap(IChild)) THEN
                    Exchange(aHeap(IParent), aHeap(IChild));
                END IF;

                Placed := True;

            END IF;

        END LOOP;

    END AlmostHeapToHeap;

END GenericHeap;

```



## --Contents of GenericPriorityQueue.ads

GENERIC

```
TYPE KeyType IS PRIVATE;
TYPE ElementType IS PRIVATE;
WITH FUNCTION KeyOf (Element: ElementType) RETURN KeyType IS <>;
WITH FUNCTION "<"(Left, Right: KeyType) RETURN Boolean IS <>;
with procedure MyPut(x: KeyType);
```

PACKAGE GenericPriorityQueue IS

```
-----
-
--| Generic package for Priority Queues
--| "<" is used as the means of assigning priority;
--| "<" means lower priority
--| Author:
--| Last Modified: September 13, 2002
-----
-

-- type definition

TYPE Queue (Capacity: Positive) IS LIMITED PRIVATE;

-- exported exceptions

QueueFull  : EXCEPTION;
QueueEmpty : EXCEPTION;

-- constructors

PROCEDURE MakeEmpty (Q : IN OUT Queue);
-- Pre:      Q is defined
-- Post:      Q is empty

PROCEDURE Enqueue (Q : IN OUT Queue; E : IN ElementType);
-- Pre:      Q and E are defined
-- Post:      Q is returned with E inserted in its proper
--            position according to Smaller: the largest Element is at
--            the head of the queue.
-- Raises: QueueFull if Q already contains Capacity Elements

PROCEDURE Dequeue (Q : IN OUT Queue);
-- Pre:      Q is defined
-- Post:      Q is returned with the first Element discarded
-- Raises: QueueEmpty if Q contains no Elements

-- selector

FUNCTION First (Q : IN Queue) RETURN ElementType;
-- Pre:      Q is defined
-- Post:      The first Element of Q is returned
-- Raises: QueueEmpty if Q contains no Elements

-- inquiry operations
```

```

FUNCTION IsEmpty (Q : IN Queue) RETURN Boolean;
-- Pre:      Q is defined
-- Post:     returns True if Q is empty, False otherwise

FUNCTION IsFull  (Q : IN Queue) RETURN Boolean;
-- Pre:      Q is defined
-- Post:     returns True if Q is full, False otherwise

PRIVATE

TYPE List IS ARRAY (Positive RANGE <>) OF ElementType;
TYPE Queue (Capacity: Positive) IS RECORD
    CurrentSize: Natural := 0;
    Store       : List(1..Capacity);
END RECORD;

END GenericPriorityQueue;

--Contents of GenericPriorityQueue.adb
with GenericHeap;
package body GenericPriorityQueue is
-----
-
--| Body of Generic Priority Queue Package
--| Author:
--| Last Modified: September 13, 2002
-----
-

-- instantiate generic heap package for these conditions
package Heaps is
    new GenericHeap(KeyType => KeyType,
                    ElementType => ElementType,
                    IndexType => Positive,
                    ListType => List,
                    KeyOf => KeyOf,
                    "<" => "<",
                    MyPut => MyPut
    );

procedure MakeEmpty (Q : IN OUT Queue) is
begin
    Q.CurrentSize := 0;
end MakeEmpty;

procedure Enqueue (Q : IN OUT Queue; E : IN ElementType) is
begin
    if IsFull(Q) then
        RAISE QueueFull; -- Generate interrupt
    else
        -- put new item at end of heap, then move it up the tree.
        Q.CurrentSize := Q.CurrentSize + 1;
        Q.Store (Q.CurrentSize) := E;
        Heaps.ExtendHeap(Q.Store(1..Q.CurrentSize));
    end if;
end Enqueue;

```

```

end Enqueue;

procedure Dequeue (Q : IN OUT Queue) is
begin
  if IsEmpty (Q) then
    RAISE QueueEmpty;
  else
    -- Replace first item in heap with the last item,
    -- decrease heap size by 1, then move the new top item
    -- down the heap to its proper position.
    Q.Store(1) := Q.Store(Q.CurrentSize);
    Q.CurrentSize := Q.CurrentSize - 1;
    Heaps.AlmostHeapToHeap(Q.Store(1..Q.CurrentSize));
  end if;
end Dequeue;

function First (Q : IN Queue) return ElementType is
begin
  if IsEmpty(Q) then
    RAISE QueueEmpty;
  else
    return Q.Store (1);
  end if;
end First;

function IsEmpty (Q : IN Queue) return Boolean is
begin
  return Q.CurrentSize = 0;
end IsEmpty;

function IsFull (Q : IN Queue) return Boolean is
begin
  return Q.CurrentSize = Q.Capacity;
end IsFull;

end GenericPriorityQueue;

--Contents of UseGenericPriorityQueue.adb
-----
-- This demonstrate the use of a heap as a priority queue. The Name field
-- of type StudentName in records of type StudentMonthOfBirth is used as
-- the key field. The larger the name (lexicographically), the higher
-- the priority is assumed to be. Note we place four name in the queue then
-- pop them out. The result is effectively a sort in descending key order.

-- The procedure MYPut(X: StudentName) is shown only to demonstrate the syntax
-- for passing an I/O routine from the main program to a generic.
-----

with GenericPriorityQueue, Ada.Text_IO; use Ada.Text_IO;

procedure UseGenericPriorityQueue is
  type MonthName is (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct,
Nov, Dec);
  type StudentName is (Adam, Candy, Don, Ellen, Evan, Flow, Glenn,
Mary, Robin, Sara, Tom, Will);

```

```

type StudentMonthOfBirth is record
    Name:      StudentName;
    MonBirth:  MonthName;
end record;

package MonthNameIO is new Ada.Text_IO Enumeration_IO (MonthName);
Package StudentNameIO is new
Ada.Text_IO Enumeration_IO (StudentName);
use MonthNameIO, StudentNameIO;

procedure MyPut(x: StudentName) is begin put(x); end;  --debug only

function GetKey( StuRec: StudentMonthOfBirth ) return StudentName is
begin
    return StuRec.Name;
end;

function "<" ( arg1, arg2: StudentName ) return Boolean is
begin
    if StudentName'pos(arg1) < StudentName'pos(arg2) then
        return true;
    else
        return false;
    end if;
end;

package GenPriQueue is new GenericPriorityQueue( StudentName,
  StudentMonthOfBirth, GetKey, "<", MyPut);

use GenPriQueue;

StudentPriQueue: Queue(6);
Student: StudentMonthOfBirth;

begin
    MakeEmpty( StudentPriQueue);
    Student.Name := Mary;
    Student.MonBirth := Jan;
    Enqueue(StudentPriQueue, Student);

    Student.Name := Adam;
    Student.MonBirth := Mar;
    Enqueue(StudentPriQueue, Student);

    Student.Name := Evan;
    Student.MonBirth := Dec;
    Enqueue(StudentPriQueue, Student);

    Student.Name := Robin;
    Student.MonBirth := Oct;
    Enqueue(StudentPriQueue, Student);

    Student.Name := Flow;
    Student.MonBirth := Oct;
    Enqueue(StudentPriQueue, Student);

```

```

while( NOT( IsEmpty(StudentPriQueue) ) ) loop
    Student := First(StudentPriQueue);
    put(Student.Name); put ( " "); put(Student.MonBirth); new_line;
    Dequeue( StudentPriQueue );
end loop;

-- Interrupt handling code for emergencies only.
exception
when QueueFull =>
    put("Tried to insert in a full queue");
when QueueEmpty =>
    put("Tried to delete from empty queue");
when Numeric_Error | Constraint_Error =>
    put("numeric or constraint error");
when Storage_Error =>
    put("Storage error");
when others =>
    put("Something went wrong");

end UseGenericPriorityQueue;

-- Output (all uppercase due to default selected in Ada.Text_IO)
ROBIN OCT
MARY JAN
FLOW OCT
EVAN DEC
ADAM MAR

```

## 10.1 The Package Text\_IO

Static Semantics

The library package Text\_IO has the following declaration:

```
with Ada.IO_Exceptions;
package Ada.Text_IO is
  type File_Type is limited private;
  type File_Mode is (In_File, Out_File, Append_File);
  type Count is range 0 .. implementation-defined;
  subtype Positive_Count is Count range 1 .. Count'Last;
  Unbounded : constant Count := 0; -- line and page length
  subtype Field is Integer range 0 .. implementation-defined;
  subtype Number_Base is Integer range 2 .. 16;
  type Type_Set is (Lower_Case, Upper_Case);
  -- File Management
  procedure Create (File : in out File_Type;
    Mode : in File_Mode := Out_File;
    Name : in String := "";
    Form : in String := "");
  procedure Open (File : in out File_Type;
    Mode : in File_Mode;
    Name : in String;
    Form : in String := "");
  procedure Close (File : in out File_Type);
  procedure Delete (File : in out File_Type);
  procedure Reset (File : in out File_Type; Mode : in File_Mode);
  procedure Reset (File : in out File_Type);
  function Mode (File : in File_Type) return File_Mode;
  function Name (File : in File_Type) return String;
  function Form (File : in File_Type) return String;
  function Is_Open (File : in File_Type) return Boolean;
  -- Control of default input and output files
  procedure Set_Input (File : in File_Type);
  procedure Set_Output (File : in File_Type);
  procedure Set_Error (File : in File_Type);
  function Standard_Input return File_Type;
  function Standard_Output return File_Type;
  function Standard_Error return File_Type;
  function Current_Input return File_Type;
  function Current_Output return File_Type;
  function Current_Error return File_Type;
  type File_Access is access constant File_Type;
  function Standard_Input return File_Access;
  function Standard_Output return File_Access;
  function Standard_Error return File_Access;
  function Current_Input return File_Access;
  function Current_Output return File_Access;
  function Current_Error return File_Access;
  -- Buffer control
  procedure Flush (File : in out File_Type);
  procedure Flush;
```

-- Specification of line and page lengths

```
procedure Set_Line_Length (File : in File_Type; To : in Count);
procedure Set_Line_Length (To : in Count);
procedure Set_Page_Length (File : in File_Type; To : in Count);
procedure Set_Page_Length (To : in Count);
function Line_Length (File : in File_Type) return Count;
function Line_Length return Count;
function Page_Length (File : in File_Type) return Count;
function Page_Length return Count;
```

-- Column, Line, and Page Control

```
procedure New_Line (File : in File_Type;
  Spacing : in Positive_Count := 1);
procedure New_Line (Spacing : in Positive_Count := 1);
procedure Skip_Line (File : in File_Type;
  Spacing : in Positive_Count := 1);
procedure Skip_Line (Spacing : in Positive_Count := 1);
function End_Of_Line (File : in File_Type) return Boolean;
function End_Of_Line return Boolean;
procedure New_Page (File : in File_Type);
procedure New_Page;
procedure Skip_Page (File : in File_Type);
procedure Skip_Page;
function End_Of_Page (File : in File_Type) return Boolean;
function End_Of_Page return Boolean;
function End_Of_File (File : in File_Type) return Boolean;
function End_Of_File return Boolean;
procedure Set_Col (File : in File_Type; To : in Positive_Count);
procedure Set_Col (To : in Positive_Count);
procedure Set_Line (File : in File_Type; To : in Positive_Count);
procedure Set_Line (To : in Positive_Count);
function Col (File : in File_Type) return Positive_Count;
function Col return Positive_Count;
function Line (File : in File_Type) return Positive_Count;
function Line return Positive_Count;
function Page (File : in File_Type) return Positive_Count;
function Page return Positive_Count;
```

-- Character Input-Output

```
procedure Get (File : in File_Type; Item : out Character);
procedure Get (Item : out Character);
procedure Put (File : in File_Type; Item : in Character);
procedure Put (Item : in Character);
procedure Look_Ahead (File : in File_Type;
  Item : out Character;
  End_Of_Line : out Boolean);
procedure Look_Ahead (Item : out Character;
  End_Of_Line : out Boolean);
procedure Get_Immediate (File : in File_Type;
  Item : out Character);
procedure Get_Immediate (Item : out Character);
procedure Get_Immediate (File : in File_Type;
  Item : out Character;
  Available : out Boolean);
procedure Get_Immediate (Item : out Character;
  Available : out Boolean);
```

-- String Input-Output

```
procedure Get (File : in File_Type; Item : out String);
procedure Get (Item : out String);
```

```

procedure Put(File : in File_Type; Item : in String);
procedure Put(Item : in String);
procedure Get_Line(File : in File_Type;
                  Item : out String;
                  Last : out Natural);
procedure Get_Line(Item : out String; Last : out Natural);
procedure Put_Line(File : in File_Type; Item : in String);
procedure Put_Line(Item : in String);

```

-- Generic packages for Input-Output of Integer Types

```

generic
  type Num is range <>;
package Integer_IO is
  Default_Width : Field := Num'Width;
  Default_Base : Number_Base := 10;
  procedure Get(File : in File_Type;
                Item : out Num;
                Width : in Field := 0);
  procedure Get(Item : out Num;
                Width : in Field := 0);
  procedure Put(File : in File_Type;
                Item : in Num;
                Width : in Field := Default_Width;
                Base : in Number_Base := Default_Base);
  procedure Put(Item : in Num;
                Width : in Field := Default_Width;
                Base : in Number_Base := Default_Base);
  procedure Get(From : in String;
                Item : out Num;
                Last : out Positive);
  procedure Put(To : out String;
                Item : in Num;
                Base : in Number_Base := Default_Base);
end Integer_IO;

```

```

generic
  type Num is mod <>;
package Modular_IO is
  Default_Width : Field := Num'Width;
  Default_Base : Number_Base := 10;
  procedure Get(File : in File_Type;
                Item : out Num;
                Width : in Field := 0);
  procedure Get(Item : out Num;
                Width : in Field := 0);
  procedure Put(File : in File_Type;
                Item : in Num;
                Width : in Field := Default_Width;
                Base : in Number_Base := Default_Base);
  procedure Put(Item : in Num;
                Width : in Field := Default_Width;
                Base : in Number_Base := Default_Base);
  procedure Get(From : in String;
                Item : out Num;
                Last : out Positive);
  procedure Put(To : out String;
                Item : in Num;
                Base : in Number_Base := Default_Base);
end Modular_IO;

```

-- Generic packages for Input-Output of Real Types

```

generic
  type Num is digits <>;
package Float_IO is

```

```

  Default_Fore : Field := 2;
  Default_Aft : Field := Num'Digits-1;
  Default_Exp : Field := 3;
  procedure Get(File : in File_Type;
                Item : out Num;
                Width : in Field := 0);
  procedure Get(Item : out Num;
                Width : in Field := 0);
  procedure Put(File : in File_Type;
                Item : in Num;
                Fore : in Field := Default_Fore;
                Aft : in Field := Default_Aft;
                Exp : in Field := Default_Exp);
  procedure Put(Item : in Num;
                Fore : in Field := Default_Fore;
                Aft : in Field := Default_Aft;
                Exp : in Field := Default_Exp);
  procedure Get(From : in String;
                Item : out Num;
                Last : out Positive);
  procedure Put(To : out String;
                Item : in Num;
                Aft : in Field := Default_Aft;
                Exp : in Field := Default_Exp);
end Float_IO;

```

```

generic
  type Num is delta <>;
package Fixed_IO is
  Default_Fore : Field := Num'Fore;
  Default_Aft : Field := Num'Aft;
  Default_Exp : Field := 0;
  procedure Get(File : in File_Type;
                Item : out Num;
                Width : in Field := 0);
  procedure Get(Item : out Num;
                Width : in Field := 0);
  procedure Put(File : in File_Type;
                Item : in Num;
                Fore : in Field := Default_Fore;
                Aft : in Field := Default_Aft;
                Exp : in Field := Default_Exp);
  procedure Put(Item : in Num;
                Fore : in Field := Default_Fore;
                Aft : in Field := Default_Aft;
                Exp : in Field := Default_Exp);
  procedure Get(From : in String;
                Item : out Num;
                Last : out Positive);
  procedure Put(To : out String;
                Item : in Num;
                Aft : in Field := Default_Aft;
                Exp : in Field := Default_Exp);
end Fixed_IO;

```

```

generic
  type Num is delta <> digits <>;
package Decimal_IO is
  Default_Fore : Field := Num'Fore;
  Default_Aft : Field := Num'Aft;
  Default_Exp : Field := 0;
  procedure Get(File : in File_Type;
                Item : out Num;
                Width : in Field := 0);
  procedure Get(Item : out Num;
                Width : in Field := 0);

```

```

procedure Put(File : in File_Type;
               Item : in Num;
               Fore : in Field := Default_Fore;
               Aft  : in Field := Default_Aft;
               Exp  : in Field := Default_Exp);

procedure Put(Item : in Num;
               Fore : in Field := Default_Fore;
               Aft  : in Field := Default_Aft;
               Exp  : in Field := Default_Exp);

procedure Get(From : in String;
               Item : out Num;
               Last : out Positive);

procedure Put(To   : out String;
               Item : in Num;
               Aft  : in Field := Default_Aft;
               Exp  : in Field := Default_Exp);

end Decimal_IO;

-- Generic package for Input-Output of Enumeration Types

generic
  type Enum is (<>);
package Enumeration_IO is
  Default_Width  : Field := 0;
  Default_Setting : Type_Set := Upper_Case;

  procedure Get(File : in File_Type;
                Item : out Enum);
  procedure Get(Item : out Enum);
  procedure Put(File : in File_Type;
                Item : in Enum;
                Width : in Field := Default_Width;
                Set   : in Type_Set := Default_Setting);
  procedure Put(Item : in Enum;
                Width : in Field := Default_Width;
                Set   : in Type_Set := Default_Setting);

  procedure Get(From : in String;
                Item : out Enum;
                Last : out Positive);
  procedure Put(To   : out String;
                Item : in Enum;
                Set   : in Type_Set := Default_Setting);

end Enumeration_IO;

- Exceptions
Status_Error : exception renames IO_Exceptions.Status_Error;
Mode_Error   : exception renames IO_Exceptions.Mode_Error;
Name_Error   : exception renames IO_Exceptions.Name_Error;
Use_Error    : exception renames IO_Exceptions.Use_Error;
Device_Error : exception renames IO_Exceptions.Device_Error;
End_Error    : exception renames IO_Exceptions.End_Error;
Data_Error   : exception renames IO_Exceptions.Data_Error;
Layout_Error : exception renames IO_Exceptions.Layout_Error;

private
  ... -- not specified by the language
ad Ada.Text_IO;

```



## Modified Wirth-Weber Relations

### First

A **FIRST** B if and only if there is a rule  $A ::= B^{\circ\circ\circ}$ .

### **FIRST<sup>+</sup>**

A **FIRST<sup>+</sup>** B if and only if there is a chain of (at least one rule) rules

$$A ::= S_1^{\circ\circ\circ}, S_1 ::= S_2^{\circ\circ\circ}, \dots, S_n ::= B^{\circ\circ\circ}.$$

Note this is simply the transitive closure of **FIRST**.

**This implies A FIRST<sup>+</sup> B if and only if  $A ::=^+ B^{\circ\circ\circ}$ .**

### **FIRST<sup>\*</sup>**

A **FIRST<sup>\*</sup>** B if and only if  $A ::=^* B^{\circ\circ\circ}$ .

This relation defines the set of leading symbols, terminal or non-terminal, derivable from A in zero or more steps.

The relation FIRST<sup>+</sup> allows us to identify the head of a string.

Given:

- |    |        |                                                        |
|----|--------|--------------------------------------------------------|
| 1) | A -> B | A FIRST B, B FIRST C, C FIRST D                        |
| 2) | B -> C | A FIRST <sup>+</sup> {B, C, D}                         |
| 3) | C -> D | B FIRST <sup>+</sup> {C, D} AND C FIRST <sup>+</sup> D |

Given:

- |    |            |                                            |
|----|------------|--------------------------------------------|
| 1) | E -> E - T | E FIRST E, E FIRST T, T FIRST (, T FIRST a |
| 2) | E -> T     | E FIRST <sup>+</sup> {E, T, (, a}          |
| 3) | T -> ( E ) | T FIRST <sup>+</sup> {(, a}                |
| 4) | T -> a     |                                            |

Example given the following rules for P:

| <b>Rule of P</b>      | <b>Relation</b> | <b>R<sup>1</sup></b> |
|-----------------------|-----------------|----------------------|
| 1) <b>A -&gt; Af</b>  | A FIRST A       | A R<br>A *           |
| 2) <b>A -&gt; B</b>   | A FIRST B       | <b>A R B</b>         |
| 3) <b>B -&gt; DdC</b> | B FIRST D       | <b>B R D *</b>       |
| 4) <b>B -&gt; De</b>  | B FIRST D       | <b>B R D</b>         |
| 5) <b>C -&gt; e</b>   | C FIRST e       | <b>C R e *</b>       |
| 6) <b>D -&gt; Bf</b>  | D FIRST B       | <b>D R B *</b>       |

| <b>R<sup>2</sup></b> | <b>R<sup>3</sup> is R<sup>+</sup></b> |
|----------------------|---------------------------------------|
| <b>A R R A</b>       | <b>A R R R A</b>                      |
| <b>A R R B</b>       | <b>A R R R B</b>                      |
| <b>A R R D *</b>     | <b>A R R R D</b>                      |
| <b>B R R B *</b>     | <b>B R R R D</b>                      |
| <b>D R R D *</b>     | <b>D R R R B</b>                      |

An “\*” indicates a new element in R<sup>+</sup>.

| <b>FIRST<sup>+</sup></b> | <b>A</b> | <b>B</b>  | <b>C</b> | <b>D</b>  | <b>e</b> | <b>d</b> | <b>f</b> |
|--------------------------|----------|-----------|----------|-----------|----------|----------|----------|
| <b>A</b>                 | <b>1</b> | <b>1</b>  |          | <b>1+</b> |          |          |          |
| <b>B</b>                 |          | <b>1+</b> |          | <b>1</b>  |          |          |          |
| <b>C</b>                 |          |           |          |           | <b>1</b> |          |          |
| <b>D</b>                 |          | <b>1</b>  |          | <b>1+</b> |          |          |          |
| <b>e</b>                 |          |           |          |           |          |          |          |
| <b>d</b>                 |          |           |          |           |          |          |          |
| <b>f</b>                 |          |           |          |           |          |          |          |

A “1” indicates the relation is in FIRST, a “1+” indicates the relation is in the transitive closure.

Consider the following network. We wish to determine all nodes allowing communications specific to each node in the network. Communications are only allowed in the indicated direction (half duplex).



We obtain the following relations from the network connectivity diagram.

- 1) A First B
- 2) B First D
- 3) B First C
- 4) C First B
- 5) J First K
- 6) J First L

The relation First may be represented as a Boolean Matrix (BMR):

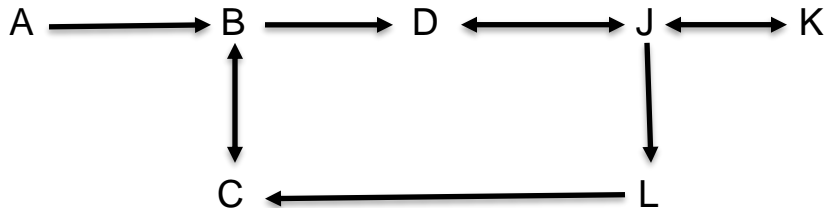
|   | A | B | C | D | J | K | L |
|---|---|---|---|---|---|---|---|
| A |   | 1 |   |   |   |   |   |
| B |   |   | 1 | 1 |   |   |   |
| C |   | 1 |   |   |   |   |   |
| D |   |   |   |   |   |   |   |
| J |   |   |   |   |   | 1 | 1 |
| K |   |   |   |   |   |   |   |
| L |   |   |   |   |   |   |   |

The transitive closure for the above BMR using Warshall's Algorithm follows:

|   | A | B  | C  | D  | J | K | L |
|---|---|----|----|----|---|---|---|
| A |   | 1  | 1' | 1' |   |   |   |
| B |   | 1' | 1  | 1  |   |   |   |
| C |   | 1  | 1' | 1' |   |   |   |
| D |   |    |    |    |   |   |   |
| J |   |    |    |    |   | 1 | 1 |
| K |   |    |    |    |   |   |   |
| L |   |    |    |    |   |   |   |

A prime has been used to indicate relations in First+. Nodes “D,” “K” and “L” may receive message but not transmit. Node “A” may transmit but not receive messages.

As a second example consider the following network.



The BMR follows for First:

|   | A | B | C | D | J | K | L |
|---|---|---|---|---|---|---|---|
| A |   | 1 |   |   |   |   |   |
| B |   |   | 1 | 1 |   |   |   |
| C |   | 1 |   |   |   |   |   |
| D |   |   |   |   | 1 |   |   |
| J |   |   |   | 1 |   | 1 | 1 |
| K |   |   |   |   |   |   | 1 |
| L |   |   | 1 |   |   |   |   |

The transitive closure of First follows . A1' (one prime) has been used to mark entries resulting from the transitive closure of First.

|   | A | B  | C  | D  | J  | K  | L  |
|---|---|----|----|----|----|----|----|
| A |   | 1  | 1' | 1' | 1' | 1' | 1' |
| B |   | 1' | 1  | 1  | 1' | 1' | 1' |
| C |   | 1  | 1' | 1' | 1' | 1' | 1' |
| D |   | 1' | 1' | 1' | 1  | 1' | 1' |
| J |   | 1' | 1' | 1  | 1' | 1  | 1  |
| K |   | 1' | 1' | 1' | 1' | 1' | 1  |
| L |   | 1' | 1  | 1' | 1' | 1' | 1' |

Note node “A” can send to all nodes but no node can transmit to node “A.” Every other node has at least one path for transmission and reception.

# Warshall's Algorithm for finding $R^+$

[discovered in 1962]

Assume an N by N Boolean Matrix Representation for the relation.

```

for I := 1, 2, ..., N loop
  for J := 1, 2, ..., N loop
    if A[J,I] = true then
      for K := 1, 2, ..., N loop
        A[J,K] := A[J,k] or A[I,K];
      end Loop;
    end if;
  end loop;
end loop;

```

## Short cut by hand:

- 1) or the  $i^{\text{th}}$  row and  $j^{\text{th}}$  row and store in row i.
- 2) Scan row i of M, if element j in row i = 1, then or row j into row i producing a new row i. Stop when an entire pass produces no new 1's for all i and j.

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix} \xRightarrow{\text{ROW 2 TO 1}} \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix} \xRightarrow{\text{ROW 3 TO 1}} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix} \xRightarrow{\text{ROW 3 TO 2}} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix} \xRightarrow{\text{ROW 1 TO 3}} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \xRightarrow{\text{ROW 1 TO 2}} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Several major companies needing help in solving a network communications problem have approached us. Essentially, the desire is given partial information about network connectivity, we have been asked to develop a method, which indicates all nodes reachable on the network from any node specified with the available information. Some network nodes are not really destinations in the traditional sense, but special nodes used to facilitate communications between nodes separated by a large distance. These nodes receive network traffic, determine network routing, and retransmit the packet towards the desired destination.

To support high traffic volume, two separate communication lines too neighboring nodes normally connect each node in the network. One line is used to receive incoming traffic (packets) from the connected node; the other line is used to transmit outgoing packets. It is possible for one or both of these communications lines to be down at any given time. Hence it is possible for a network node to be able to receive but not transmit and vice versa. In the sample network diagrams, the arrows indicate the directions in which network traffic may currently travel between nodes.

Using your knowledge of the theory of “relations ” develop a technique using BMR’s that will indicate for each node all other nodes that are reachable and all nodes that are not reachable. To receive credit, you must clearly state your algorithm then execute your algorithm using the sample network data. Clearly indicate how you use the BMR representation to determine connectivity or lack of connectivity. To satisfy management, your algorithm must be applicable to any partial network ordering.