A Practical Introduction to Data Structures and Algorithm Analysis - JAVA Edition

slides derived from material
by Clifford A. Shaffer

The Need for Data Structures

[A primary concern of this course is efficiency.]

Data structures organize data

$\Rightarrow$ **more efficient programs**. **[You might believe that faster computers make it unnecessary to be concerned with efficiency. However...]**

- More powerful computers $\Rightarrow$ more complex applications.
- YET More complex applications demand more calculations.
- Complex computing tasks are unlike our everyday experience. **[So we need special training]**

Any organization for a collection of records can be searched, processed in any order, or modified. **[If you are willing to pay enough in time delay. Ex: Simple unordered array of records.]**

- The choice of data structure and algorithm can make the difference between a program running in a few seconds or many days.

Efficiency

A solution is said to be **efficient** if it solves the problem within its **resource constraints**. **[Alt: Better than known alternatives ("relatively" efficient)]**

- space **[These are typical contraints for programs]**
- time

[This does not mean always strive for the most efficient program. If the program operates well within resource constraints, there is no benefit to making it faster or smaller.]

The **cost** of a solution is the amount of resources that the solution consumes.

Selecting a Data Structure

Select a data structure as follows:

1. Analyze the problem to determine the resource constraints a solution must meet.
2. Determine the basic operations that must be supported. Quantify the resource constraints for each operation.
3. Select the data structure that best meets these requirements.

[Typically want the “simplest” data struture that will meet requirements.]

Some questions to ask: **[These questions often help to narrow the possibilities]**

- Are all data inserted into the data structure at the beginning, or are insertions interspersed with other operations?
- Can data be deleted? **[If so, a more complex representation is typically required]**
- Are all data processed in some well-defined order, or is random access allowed?

Data Structure Philosophy

Each data structure has costs and benefits.

Rarely is one data structure better than another in all situations.

A data structure requires:

- space for each data item it stores, **[Data + Overhead]**
- time to perform each basic operation,
- programming effort. **[Some data structures/algorithms more complicated than others]**

Each problem has constraints on available space and time.

Only after a careful analysis of problem characteristics can we know the best data structure for the task.

Bank example:

- Start account: a few minutes
- Transactions: a few seconds
- Close account: overnight

Goals of this Course

1. Reinforce the concept that there are costs and benefits for every data structure. **[A worldview to adopt]**
2. Learn the commonly used data structures. These form a programmer's basic data structure "toolkit." **[The "nuts and bolts" of the course]**
3. Understand how to measure the effectiveness of a data structure or program.
   - These techniques also allow you to judge the merits of new data structures that you or others might invent. **[To prepare you for the future]**

Definitions

A **type** is a set of values.

[Ex: Integer, Boolean, Float]

A **data type** is a type and a collection of operations that manipulate the type.

[Ex: Addition]

A **data item** or **element** is a piece of information or a record.

[Physical instantiation]

A data item is said to be a **member** of a data type.

[]

A **simple data item** contains no subparts.

[Ex: Integer]

An **aggregate data item** may contain several pieces of information.

[Ex: Payroll record, city database record]

Abstract Data Types

Abstract Data Type (ADT): a definition for a data type solely in terms of a set of values and a set of operations on that data type.

Each ADT operation is defined by its inputs and outputs.

Encapsulation: hide implementation details

A **data structure** is the physical implementation of an ADT.

- Each operation associated with the ADT is implemented by one or more subroutines in the implementation.

Data structure usually refers to an organization for data in main memory.

File structure: an organization for data on peripheral storage, such as a disk drive or tape.

An ADT manages complexity through abstraction: **metaphor**. **[Hierarchies of labels]**

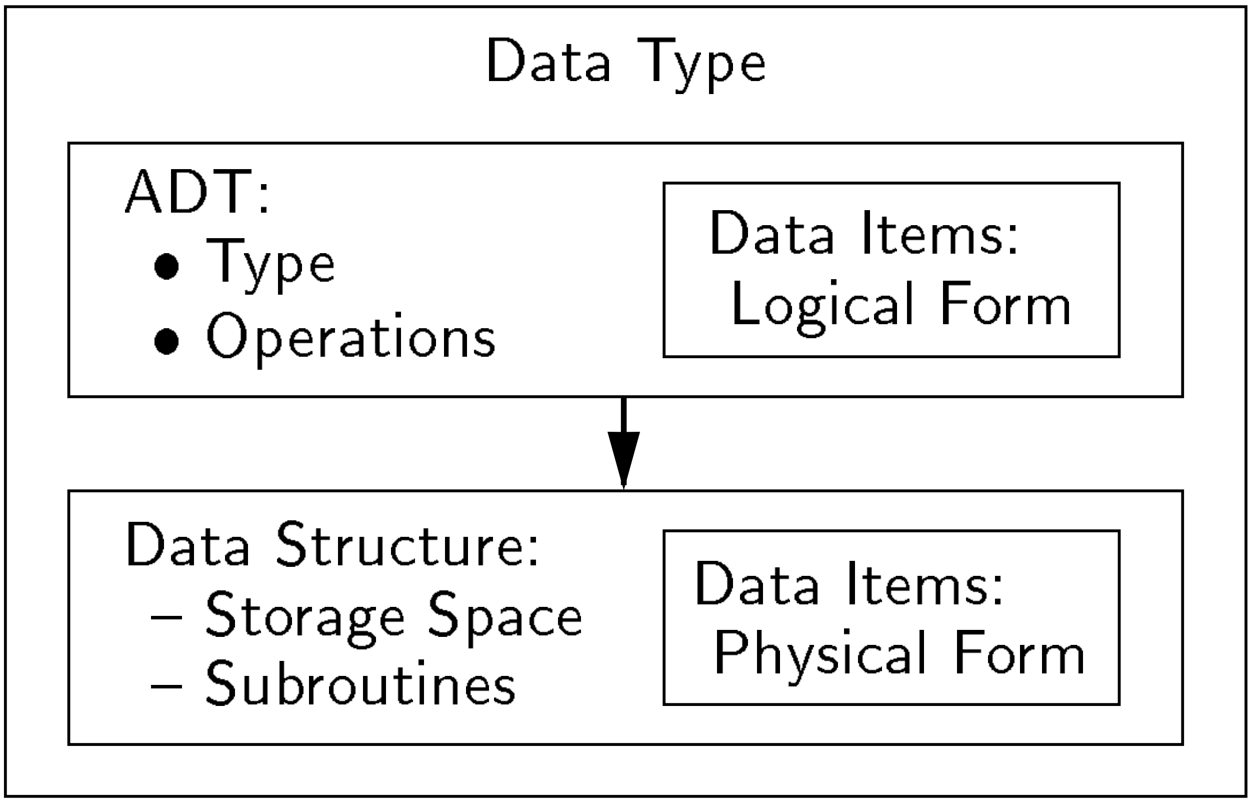
[Ex: transistors → gates → CPU. In a program, implement an ADT, then think only about the ADT, not its implementation]

Logical vs. Physical Form

Data items have both a **logical** and a **physical** form.

Logical form: definition of the data item within an ADT. **[Ex: Integers in mathematical sense: $+$, $-$]**

Physical form: implementation of the data item within a data structure. **[16/32 bit integers: overflow]**



[In this class, we frequently move above and below "the line" separating logical and physical forms.]

Problems

Problem: a task to be performed.

- Best thought of as inputs and matching outputs.
- Problem definition should include constraints on the resources that may be consumed by any acceptable solution. **[But NO constraints on HOW the problem is solved]**

Problems $\Leftrightarrow$ mathematical functions

- A **function** is a matching between inputs (the **domain**) and outputs (the **range**).
- An **input** to a function may be single number, or a collection of information.
- The values making up an input are called the **parameters** of the function.
- A particular input must always result in the same output every time the function is computed.

Algorithms and Programs

Algorithm: a method or a process followed to solve a problem. **[A recipe]**

An algorithm takes the input to a problem (function) and transforms it to the output. **[A mapping of input to output]**

A problem can have many algorithms.

An algorithm possesses the following properties:

1. It must be **correct**. **[Computes proper function]**
2. It must be composed of a series of **concrete steps**. **[Executable by that machine]**
3. There can be **no ambiguity** as to which step will be performed next.
4. It must be composed of a **finite** number of steps.
5. It must **terminate**.

A **computer program** is an instance, or concrete representation, for an algorithm in some programming language.

[We frequently interchange use of “algorithm” and “program” though they are actually different concepts]

Mathematical Background

[Look over Chapter 2, read as needed depending on your familiarity with this material.]

Set concepts and notation **[Set has no duplicates, sequence may]**

Recursion

Induction proofs

Logarithms **[Almost always use log to base 2. That is our default base.]**

Summations

Algorithm Efficiency

There are often many approaches (algorithms) to solve a problem. How do we choose between them?

At the heart of computer program design are two (sometimes conflicting) goals:

1. To design an algorithm that is easy to understand, code and debug.
2. To design an algorithm that makes efficient use of the computer's resources.

Goal (1) is the concern of Software Engineering.

Goal (2) is the concern of data structures and algorithm analysis.

When goal (2) is important, how do we measure an algorithm's cost?

How to Measure Efficiency?

1. Empirical comparison (run programs).

   [Difficult to do "fairly." Time consuming.]

2. Asymptotic Algorithm Analysis.

Critical resources:

- Time
- Space (disk, RAM)
- Programmer's effort
- Ease of use (user's effort).

Factors affecting running time:

- Machine load
- OS
- Compiler
- Problem size or Specific input values for given problem size

For most algorithms, running time depends on "size" of the input.

Running time is expressed as $\mathbf{T}(n)$ for some function $\mathbf{T}$ on input size n.

Examples of Growth Rate

Example 1: **[As n grows, how does $\mathrm{T}(n)$ grow?]**

```
static int largest(int[] array) { // Find largest val
                                  // all values >=0
  int currLargest = 0;  // Store largest val
  for (int i=0; i<array.length; i++) // For each elem
   if (array[i] > currLargest)    //   if largest
      currLargest = array[i];     //     remember it
  return currLargest;             // Return largest val
}
```

[Cost: $\mathrm{T}(n) = c_1 n + c_2$ steps]

Example 2: Assignment statement **[Constant cost]**

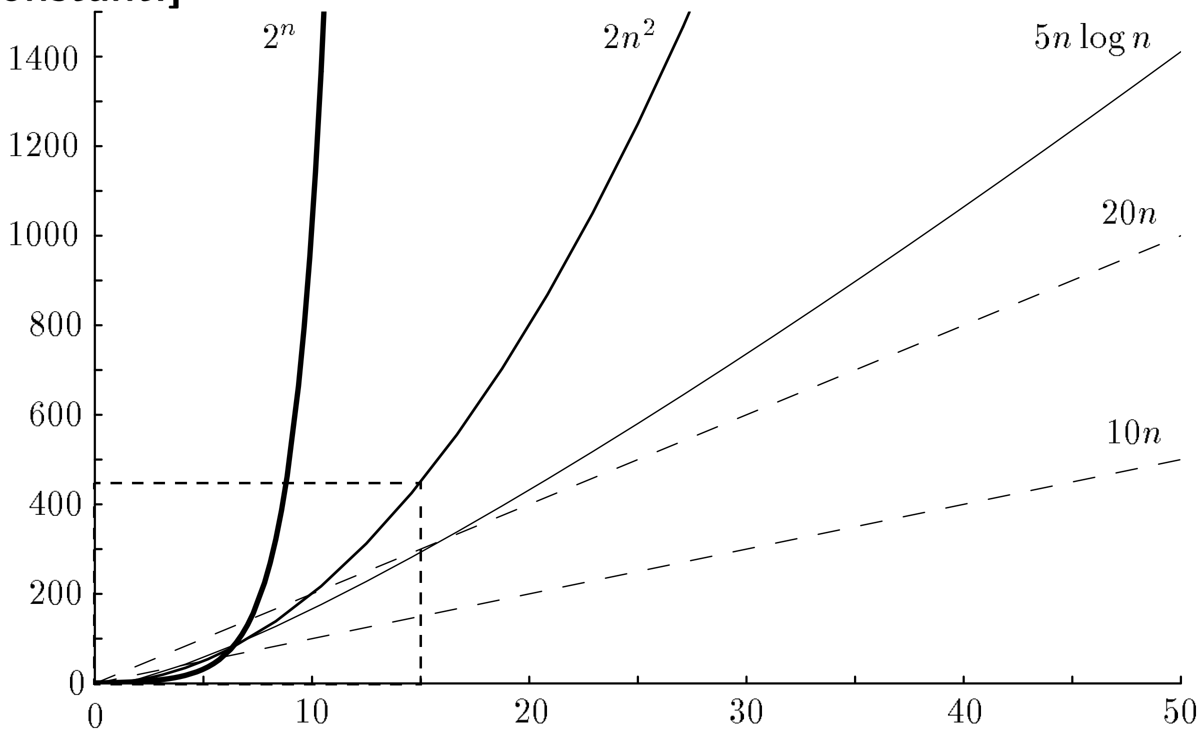
Example 3:

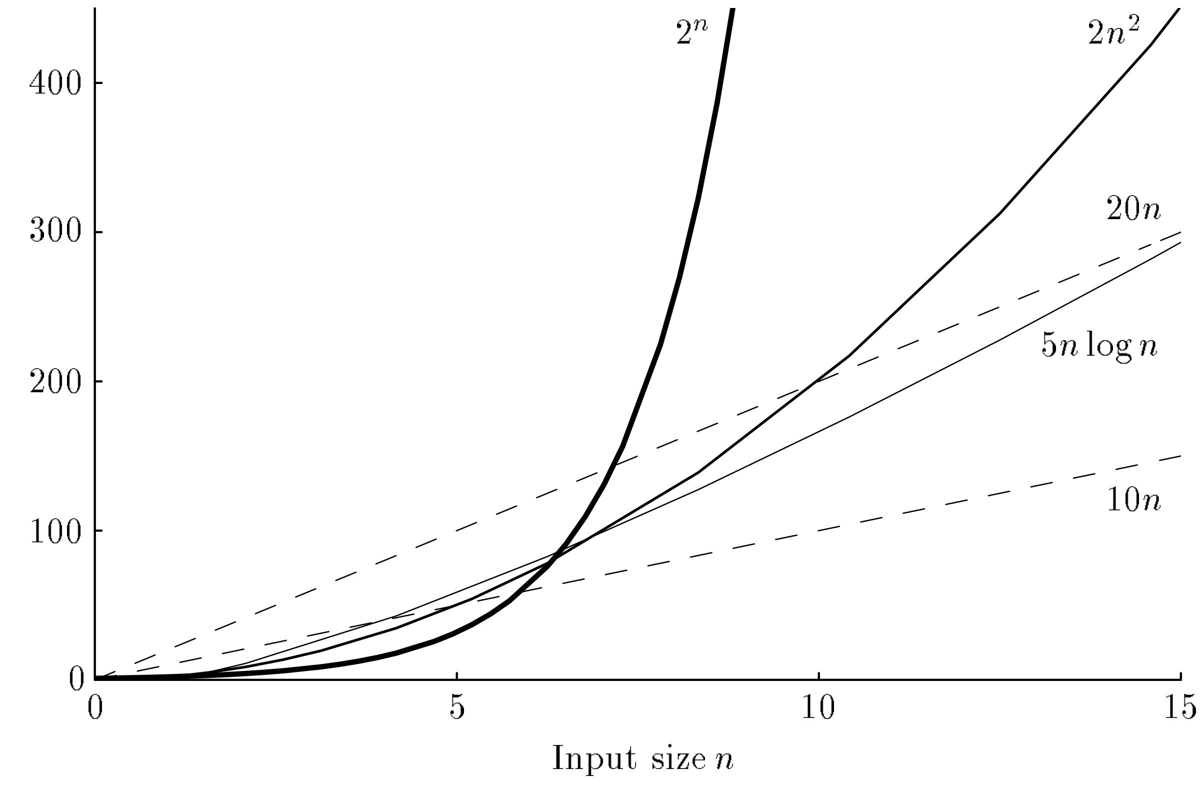
```
sum = 0;
for (i=1; i<=n; i++)
   for (j=1; j<=n; j++)
      sum++;
```

[Cost: $\mathrm{T}(n) = c_1 n^2 + c_2$ Roughly n^2 steps, with sum being n^2 at the end. Ignore various overhead such as loop counter increments.]

Growth Rate Graph

[2^n is an exponential algorithm. $10n$ and $20n$ differ only by a constant.]





Important facts to remember

- for any integer constants $a, b > 1$ n^a grows faster than $\log^b n$

  [any polynomial is worse than any power of any logarithm]

- for any integer constants $a, b > 1$ n^a grows faster than $\log n^b$

  [any polynomial is worse than any logarithm of any power]

- for any integer constants $a, b > 1$ a^n grows faster than n^b

  [any exponential is worse than any polynomial]

Best, Worst and Average Cases

Not all inputs of a given size take the same time.

Sequential search for K in an array of n integers:

- Begin at first element in array and look at each element in turn until K is found.

Best Case: **[Find at first position: 1 compare]**

Worst Case: **[Find at last position: n compares]**

Average Case: **[$(n+1)/2$ compares]**

While average time seems to be the fairest measure, it may be difficult to determine.

[Depends on distribution. Assumption for above analysis: Equally likely at any position.]

When is worst case time important?

[algorithms for time-critical systems]

Faster Computer or Algorithm?

What happens when we buy a computer 10 times faster? **[How much speedup? 10 times. More important: How much increase in problem size for same time? Depends on growth rate.]**

$\mathrm{T}(n)$	n	n'	Change	n'/n
$10n$	$1,000$	$10,000$	$n' = 10n$	10
$20n$	500	$5,000$	$n' = 10n$	10
$5n \log n$	250	$1,842$	$\sqrt{10}n < n' < 10n$	7.37
$2n^2$	70	223	$n' = \sqrt{10}n$	3.16
2^n	13	16	$n' = n + 3$	$--$

[For n^2, if $n = 1000$, then n' would be 1003]

n: Size of input that can be processed in one hour (10,000 steps).

n': Size of input that can be processed in one hour on the new machine (100,000 steps).

[Compare $\mathrm{T}(n) = n^2$ to $\mathrm{T}(n) = n \log n$. For $n > 58$, it is faster to have the $\Theta(n \log n)$ algorithm than to have a computer that is 10 times faster.]

Asymptotic Analysis: Big-oh

Definition: For $\mathbf{T}(n)$ a non-negatively valued function, $\mathbf{T}(n)$ is in the set O$(f(n))$ if there exist two positive constants c and n_0 such that $\mathbf{T}(n) \leq cf(n)$ for all $n > n_0$.

Usage: The algorithm is in O(n^2) in [best, average, worst] case.

Meaning: For *all* data sets big enough (i.e., $n > n_0$), the algorithm *always* executes in less than $cf(n)$ steps [in best, average or worst case].

[Must pick one of these to complete the statement. Big-oh notation applies to some set of inputs.]

Upper Bound.

Example: if $\mathbf{T}(n) = 3n^2$ then $\mathbf{T}(n)$ is in O(n^2).

Wish tightest upper bound:
While $\mathbf{T}(n) = 3n^2$ is in O(n^3), we prefer O(n^2).

[It provides more information to say O(n^2) than O(n^3)]

Big-oh Example

Example 1. Finding value X in an array. **[Average case]**

$\mathbf{T}(n) = c_s n/2$. **[c_s is a constant. Actual value is irrelevant]**
For all values of $n > 1$, $c_s n/2 \leq c_s n$.
Therefore, by the definition, $\mathbf{T}(n)$ is in O(n) for $n_0 = 1$ and $c = c_s$.

Example 2. $\mathbf{T}(n) = c_1 n^2 + c_2 n$ in average case

$c_1 n^2 + c_2 n \leq c_1 n^2 + c_2 n^2 \leq (c_1 + c_2) n^2$ for all $n > 1$.

$\mathbf{T}(n) \leq c n^2$ for $c = c_1 + c_2$ and $n_0 = 1$.

Therefore, $\mathbf{T}(n)$ is in O(n^2) by the definition.

Example 3: $\mathbf{T}(n) = c$. We say this is in O(1). **[Rather than O(c)]**

Big-Omega

Definition: For $\mathbf{T}(n)$ a non-negatively valued function, $\mathbf{T}(n)$ is in the set $\Omega(g(n))$ if there exist two positive constants c and n_0 such that $\mathbf{T}(n) \geq cg(n)$ for all $n > n_0$.

Meaning: For *all* data sets big enough (i.e., $n > n_0$), the algorithm *always* executes in more than $cg(n)$ steps.

Lower Bound.

Example: $\mathbf{T}(n) = c_1n^2 + c_2n$.

$c_1n^2 + c_2n \geq c_1n^2$ for all $n > 1$.
$\mathbf{T}(n) \geq cn^2$ for $c = c_1$ and $n_0 = 1$.

Therefore, $\mathbf{T}(n)$ is in $\Omega(n^2)$ by the definition.

Want greatest lower bound.

Theta Notation

When big-Oh and Ω meet, we indicate this by using Θ (big-Theta) notation.

Definition: An algorithm is said to be $\Theta(h(n))$ if it is in $O(h(n))$ and it is in $\Omega(h(n))$.

[For polynomial equations on $\mathrm{T}(n)$, we always have Θ. There is no uncertainty, a "complete" analysis.]

Simplifying Rules:

1. If $f(n)$ is in $O(g(n))$ and $g(n)$ is in $O(h(n))$, then $f(n)$ is in $O(h(n))$.
2. If $f(n)$ is in $O(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $O(g(n))$. **[No constant]**
3. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $(f_1 + f_2)(n)$ is in $O(\max(g_1(n), g_2(n)))$. **[Drop low order terms]**
4. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$ then $f_1(n)f_2(n)$ is in $O(g_1(n)g_2(n))$. **[Loops]**

Running Time of a Program

[Asymptotic analysis is defined for equations. Need to convert program to an equation.]

Example 1: `a = b;`

This assignment takes constant time, so it is $\Theta(1)$. **[Not $\Theta(c)$ – notation by tradition]**

Example 2:

```
sum = 0;
for (i=1; i<=n; i++)
   sum += n;
```

[$\Theta(n)$ (even though sum is n^2)]

Example 3:

```
sum = 0;
for (j=1; j<=n; j++)      // First for loop
   for (i=1; i<=j; i++)  //   is a double loop
      sum++;
for (k=0; k<n; k++)      // Second for loop
   A[k] = k;
```

[First statement is $\Theta(1)$. Double for loop is $\sum i = \Theta(n^2)$. Final for loop is $\Theta(n)$. Result: $\Theta(n^2)$.]

More Examples

Example 4.

```
sum1 = 0;
for (i=1; i<=n; i++)      // First double loop
   for (j=1; j<=n; j++)  //   do n times
      sum1++;

sum2 = 0;
for (i=1; i<=n; i++)      // Second double loop
   for (j=1; j<=i; j++)  //   do i times
      sum2++;
```

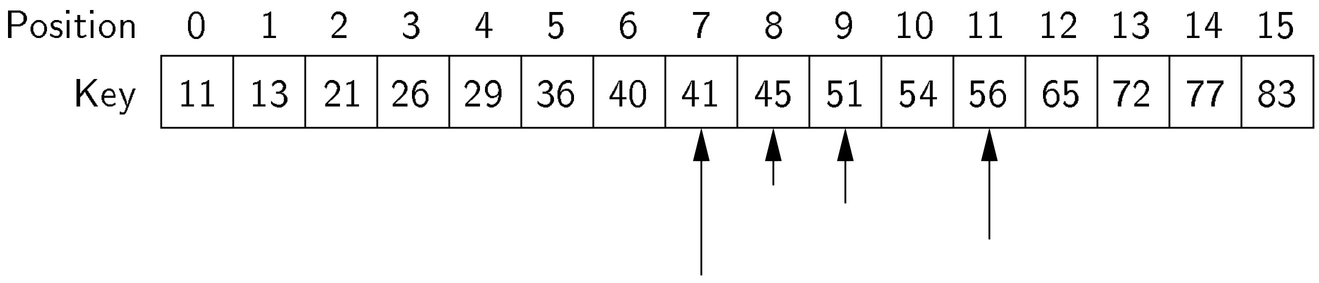
[First loop, sum is n^2. Second loop, sum is $(n+1)(n)/2$. Both are $\Theta(n^2)$.]

Example 5.

```
sum1 = 0;
for (k=1; k<=n; k*=2)
   for (j=1; j<=n; j++)
      sum1++;

sum2 = 0;
for (k=1; k<=n; k*=2)
   for (j=1; j<=k; j++)
      sum2++;
```

[First is $\sum_{k=1}^{\log n} n = \Theta(n \log n)$. Second is $\sum_{k=0}^{\log n - 1} 2^k = \Theta(n)$.]

Binary Search



Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Key	11	13	21	26	29	36	40	41	45	51	54	56	65	72	77	83

```
static int binary(int K, int[] array,
                         int left, int right) {
  // Return position in array (if any) with value K
  int l = left-1;
  int r = right+1;   // l and r are beyond array bounds
                     // to consider all array elements
  while (l+1 != r) { // Stop when l and r meet
    int i = (l+r)/2; // Look at middle of subarray
    if (K < array[i]) r = i;     // In left half
    if (K == array[i]) return i; // Found it
    if (K > array[i]) l = i;     // In right half
  }
  return UNSUCCESSFUL; // Search value not in array
}
```

invocation of binary

int pos = binary(43, ar, 0, 15);

Analysis: How many elements can be examined in the worst case? **[$\Theta(\log n)$]**

Other Control Statements

`while` loop: analyze like a `for` loop.

`if` statement: Take greater complexity of then/else clauses.

[If probabilities are independent of n.]

`switch` statement: Take complexity of most expensive case.

[If probabilities are independent of n.]

Subroutine call: Complexity of the subroutine.

Analyzing Problems

Use same techniques to analyze problems, i.e. any possible algorithm for a given problem (e.g., sorting)

Upper bound: Upper bound of best known algorithm.

Lower bound: Lower bound for *every possible algorithm*.

[The examples so far have been easy in that exact equations always yield Θ. Thus, it was hard to distinguish Ω and O. Following example should help to explain the difference – bounds are used to describe our level of uncertainty about an algorithm.]

Example: Sorting

1. Cost of I/O: $\Omega(n)$
2. Bubble or insertion sort: $O(n^2)$
3. A better sort (Quicksort, Mergesort, Heapsort, etc.): $O(n \log n)$
4. We prove later that sorting is $\Omega(n \log n)$

Multiple Parameters

[Ex: 256 colors (8 bits), 1000×1000 **pixels]**

Compute the rank ordering for all C (256) pixel values in a picture of P pixels.

```
for (i=0; i<C; i++)   // Initialize count
   count[i] = 0;
for (i=0; i<P; i++)   // Look at all of the pixels
   count[value(i)]++; // Increment proper value count
sort(count);          // Sort pixel value counts
```

If we use P as the measure, then time is $\Theta(P \log P)$.

But this is wrong because we sort $colors$
More accurate is $\Theta(P + C \log C)$.
If $C << P$, P could overcome $C \log C$

Space Bounds

Space bounds can also be analyzed with asymptotic complexity analysis.

Time: Algorithm
Space: Data Structure

Space/Time Tradeoff Principle:
One can often achieve a reduction in time is one is willing to sacrifice space, or vice versa.

- Encoding or packing information
  Boolean flags
- Table lookup
  Factorials

Disk Based Space/Time Tradeoff Principle:
The smaller you can make your disk storage requirements, the faster your program will run.
(because access to disk is typically more costly than "any" computation)

Algorithm Design methods: Divide et impera

Decompose a problem of size n into (one or more) problems of size $m < n$

Solve subproblems, if reduced size is not "trivial", in the same manner, possibly combining solutions of the subproblems to obtain the solution of the original one ...

... until size becomes "small enough" (typically 1 or 2) to solve the problem directly (without decomposition)

Complexity can be typically analyzed by means of **recurrence equations**

Recurrence Equations(1)

we have already seen the following

$\mathsf{T}(n) = a\mathsf{T}(n/b) + cn^k$, for $n > 1$
$\mathsf{T}(1) = \mathsf{d}$,

Solution of the recurrence depends on the ratio $r = b^k/a$

$\mathsf{T}(n) = \Theta(n^{\log_b a})$, if $a > b^k$
$\mathsf{T}(n) = \Theta(n^k \log n)$, if $a = b^k$
$\mathsf{T}(n) = \Theta(n^k)$, if $a < b^k$

Complexity depends on

- relation between a and b, i.e., whether all subproblems need to be solved or only some do
- value of k, i.e., amount of additional work to be done to partition into subproblems and combine solutions

Recurrence Equations(2)

Examples

- $a = 1, b = 2$ (two halves, solve only one), $k = 0$ (constant partition+combination overhead): e.g., Binary search: $\mathsf{T}(n) = \Theta(\log n)$ (extremely efficient!)
- $a = b = 2$ (two halves) and (k=1) (partitioning+combination $\Theta(n)$) $\mathsf{T}(n) = \Theta(n \log n)$; e.g., Mergesort;
- $a = b$ (partition data and solve for all partitions) and $k = 0$ (constant partition+combining) $\mathsf{T}(n) = \Theta(n^{\log_b a}) = \Theta(n)$, same as linear/sequential processing (E.g., finding the max/min element in an array)

Now we'll see

1. max/min search as an example of linear complexity
2. other kinds of recurrence equations
   - $\mathsf{T}(n)$=$\mathsf{T}(n-1)$+n leads to **quadratic** complexity: example bubblesort;
   - $\mathsf{T}(n)$=$a\mathsf{T}(n-1)$+k leads to **exponential** complexity: example Towers of Hanoi

MaxMin search(1)

"Obvious" method: sequential search

```
public class MinMaxPair {
   public int min;
   public int max;
}

public static MinMaxPair minMax (float [] a) {
   //guess a[0] as min and max
   MaxMinPair p = new MaxMinPair(); p.min = p.max = 0;
   // search in the remaining part of the array
   for (int i = 1; i<a.length; i++) {
      if (a[i]<a[p.min]) p.min = i;
      if (a[i]>a[p.max]) p.max = i;
   }
   return p;
}
```

Complexity is $\mathrm{T}(n)=2(n-1)=\Theta(n)$

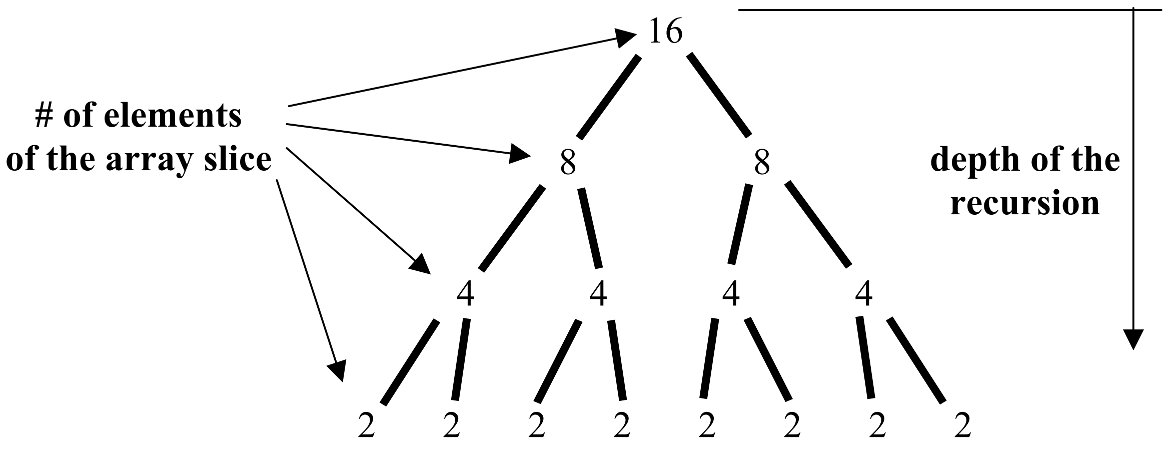
Divide et impera approach: split array in two, find MinMax of each, choose overall min among the two mins and max among the two maxs

MaxMin search(2)

```
public static MinMaxPair minMax (float [] a,
                                 int l, int r) {
   MaxMinPair p = new MinMaxPair();
   if(l==r) {p.min = p.max = r; return p;}
   if (l==r-1) {
     if (a[l]<a[r]) {
       p.min=l; p.max=r;
     }
     else {
     p.min=r; p.max=l;
     }
     return p;
   }
   int m = (l+r)/2;
   MinMaxPair p1 = minMax(a, l, m);
   MinMaxPair p2 = minMax(a, m+1, r);
   if (a[p1.min]<a[p2.min]) p.min=p1.min else p.min=p2.min;
   if (a[p1.max]>a[p2.max]) p.max=p1.max else p.max=p2.max;
   return p;
}
```

Asymptotic complexity analyzable by means of recurrence
$\mathsf{T}(n) = a\mathsf{T}(n/b) + cn^k$, for $n > 1$
$\mathsf{T}(1) = \mathsf{d}$,

We have $a = b$ and $k = 0$ hence $\mathsf{T}(n) = \Theta(n)$, apparently no improvement: we need a more precise analysis



Assume for simplicity n is a power of 2. Here is the tree of recursive calls for $n = 16$. There are

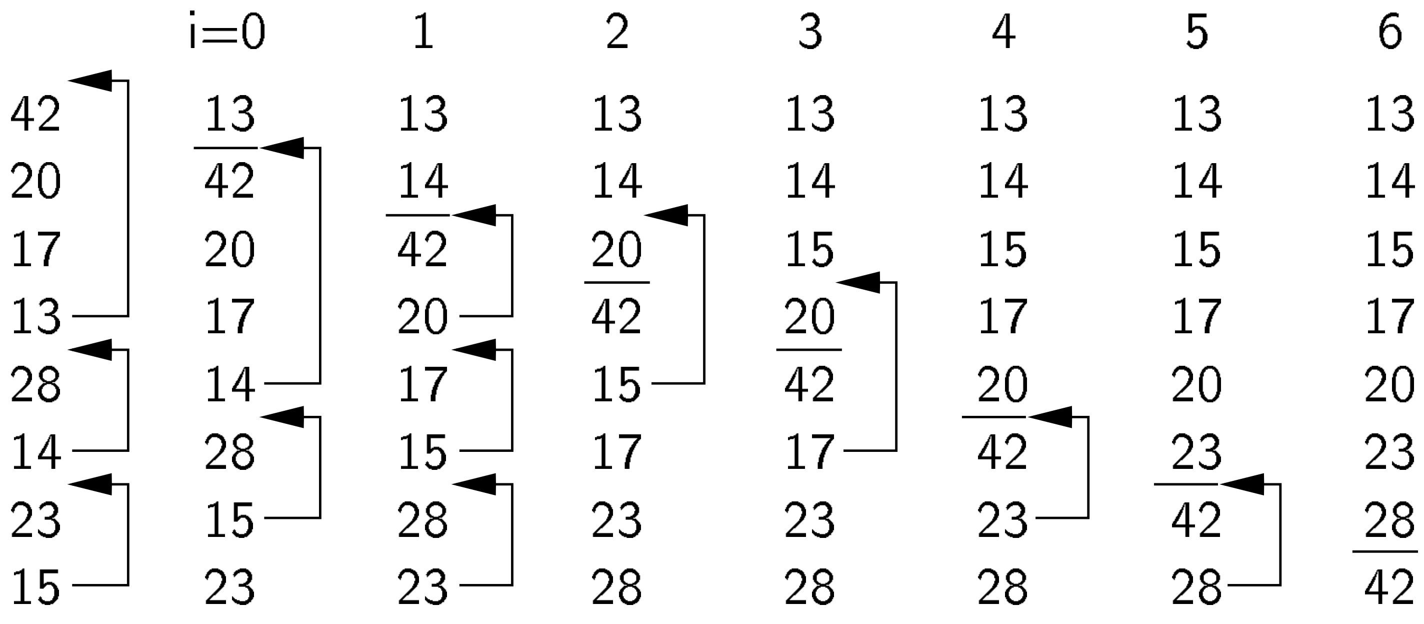
- $n/2$ leaf nodes, each of which takes 1 comparison
- $n/2 - 1$ internal nodes each of which takes 2 comparison
- hence #comparisons = $2(n/2 - 1) + n/2 = (3/2)n - 2$, a 25% improvement wrt linear search

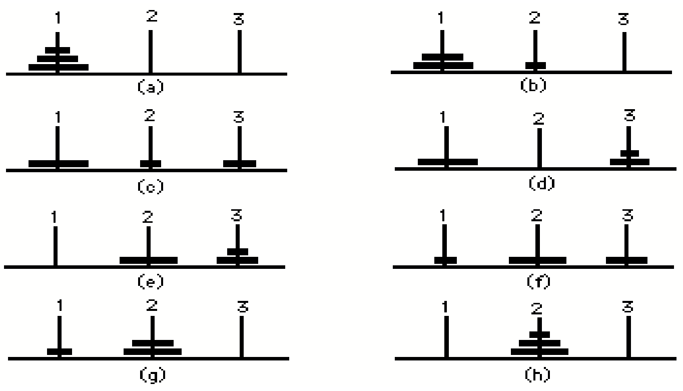
bubblesort as a divide et impera algorithm

To sort an array of n element, put the smallest element in first position, then sort the remaining part of the array.

Putting the smallest element to first position requires an array traversal ($\Theta(n)$ complexity)

```
static void bubsort(Elem[] array) {    // Bubble Sort
  for (int i=0; i<array.length-1; i++) // Bubble up
    //take i-th smallest to i-th place
    for (int j=array.length-1; j>i; j--)
      if (array[j].key() < array[j-1].key())
        DSutil.swap(array, j, j-1);
}
```





Move stack of rings form one pole to another, with following constraints

- move one ring at a time
- never place a ring on top of a smaller one

Divide et impera approach: move stack of $n-1$ smaller rings on third pole as a support, then move largest ring, then move stack of $n-1$ smaller rings from support pole to destination pole using start pole as a support

```
static void TOH(int n,
                Pole start, Pole goal, Pole temp) {
  if (n==1) System.out.println("move ring from pole " +
            + start + " to pole " + goal);
  else {
   TOH(n-1, start, temp, goal);
   System.out.println("move ring from pole " +
                     + start + " to pole " + goal);
   TOH(n-1, temp, goal, start);
  }
}
```

Time complexity as a function of the size n of the ring stack: T(n)=2^n-1

Exponential complexity of Towers of Hanoi

Recurrence equation is T(n)=2T($n-1$)+1 for $n > 1$, and T(1)=1.

A special case of the more general recurrence T(n)=aT($n-1$)+k, for $n > 1$, and T(1)=k.

It is easy to show that the solution is T(n)=$k\sum_{i=0}^{n-1} a^i$ hence T(n)=$\Theta(a^n)$

Why? A simple proof by induction.

Base: T(1)=k= $k\sum_{i=0}^{0} a^i$

Induction:

T($n+1$)=aT(n)+k=

$=ak\sum_{i=0}^{n-1} a^i + k = k\sum_{i=1}^{n} a^i + k = k\sum_{i=0}^{n} a^i=$

$=k\sum_{i=0}^{(n+1)-1} a^i$

In the case of Towers of Hanoi $a = 2, k = 1$, hence T(n)=$\sum_{i=0}^{n-1} 2^i = 2^n$-1

Lists

[Students should already be familiar with lists. Objectives: use alg analysis in familiar context, compare implementations.]

A **list** is a finite, ordered sequence of data items called **elements**.

[The positions are ordered, NOT the values.]

Each list element has a data type.

The **empty** list contains no elements.

The **length** of the list is the number of elements currently stored.

The beginning of the list is called the **head**, the end of the list is called the **tail**.

Sorted lists have their elements positioned in ascending order of value, while **unsorted lists** have no necessary relationship between element values and positions.

Notation: (a_0, a_1, ..., a_{n-1})

What operations should we implement?

[Add/delete elem anywhere, find, next, prev, test for empty.]

List ADT

```
interface List {                        // List ADT
public void clear();                    // Remove all Objects
public void insert(Object item);        // Insert at curr pos
public void append(Object item);        // Insert at tail
public Object remove();                 // Remove/return curr
public void setFirst();                 // Set to first pos
public void next();                     // Move to next pos
public void prev();                     // Move to prev pos
public int length();                    // Return curr length
public void setPos(int pos);            // Set curr position
public void setValue(Object val);       // Set current value
public Object currValue();              // Return curr value
public boolean isEmpty();               // True if empty list
public boolean isInList();              // True if curr in list
public void print();                    // Print all elements
} // interface List
```

[This is an example of a Java interface. Any Java class using this interface must implement all of these functions. Note that the generic type "Object" is being used for the element type.]

List ADT Examples

List: (12, 32, 15)

```
MyLst.insert(element);
```

[The above is an example use of the insert function. “element” is an object of the list element data type.]

Assume `MyLst` has 32 as current element:

```
MyLst.insert(99);
```

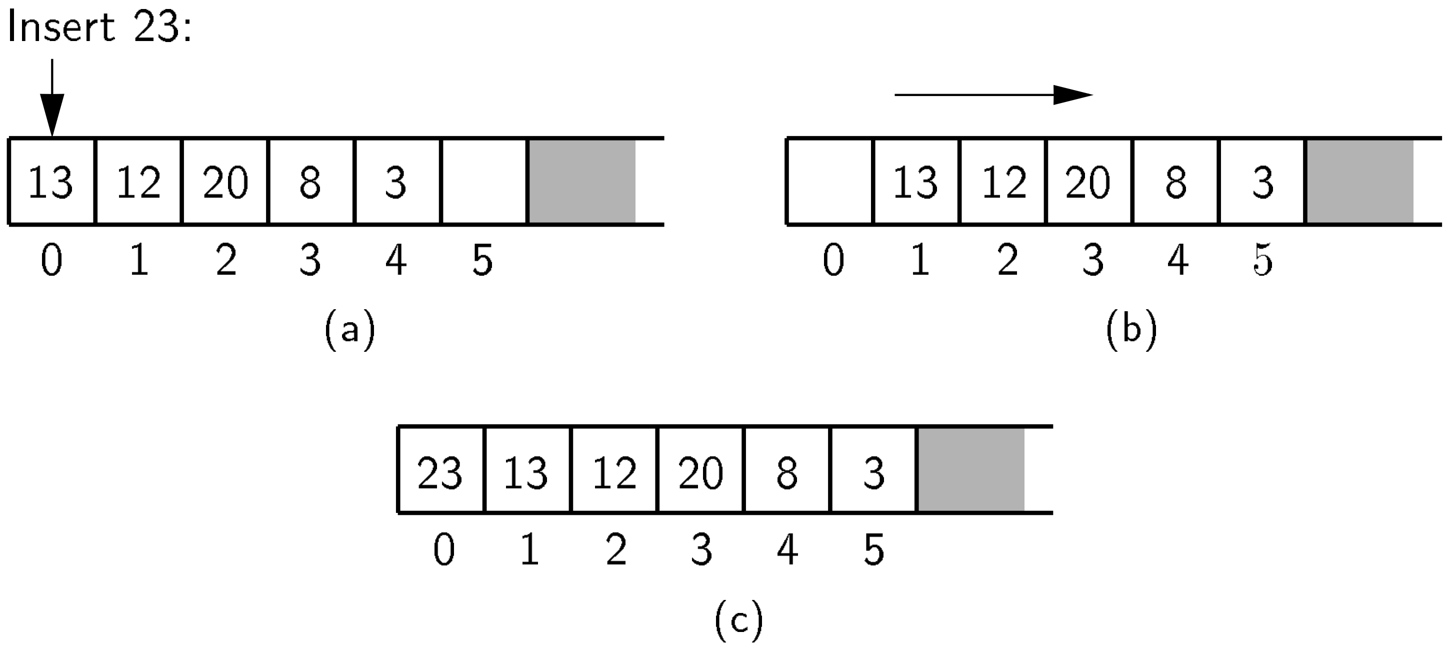
[Put 99 before current element, yielding (12, 99, 32, 15).]

Process an entire list:

```
for (MyLst.setFirst(); MyLst.isInList(); MyLst.next())
  DoSomething(MyLst.currValue());
```

Array-Based List Insert

[Push items up/down. Cost: $\Theta(n)$.]



Array-Based List Class

```
class AList implements List {   // Array-based list

private static final int defaultSize = 10;

private int msize;              // Maximum size of list
private int numInList;          // Actual list size
private int curr;               // Position of curr
private Object[] listArray;     // Array holding list

AList() { setup(defaultSize); } // Constructor
AList(int sz) { setup(sz); }    // Constructor

private void setup(int sz) {    // Do initializations
  msize = sz;
  numInList = curr = 0;
  listArray = new Object[sz];   // Create listArray
}

public void clear()     // Remove all Objects from list
{numInList = curr = 0; } // Simply reinitialize values

public void insert(Object it) { // Insert at curr pos
  Assert.notFalse(numInList < msize, "List is full");
  Assert.notFalse((curr >=0) && (curr <= numInList),
                  "Bad value for curr");
  for (int i=numInList; i>curr; i--) // Shift up
    listArray[i] = listArray[i-1];
  listArray[curr] = it;
  numInList++;                  // Increment list size
}
```

Array-Based List Class (cont)

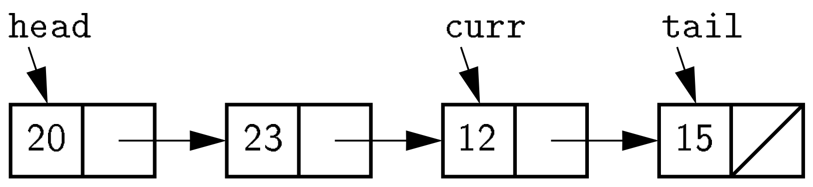
```
public void append(Object it) { // Insert at tail
  Assert.notFalse(numInList < msize, "List is full");
  listArray[numInList++] = it;  // Increment list size
}

public Object remove() { // Remove and return Object
  Assert.notFalse(!isEmpty(), "No delete: list empty");
  Assert.notFalse(isInList(), "No current element");
  Object it = listArray[curr];  // Hold removed Object
  for(int i=curr; i<numInList-1; i++) // Shift down
    listArray[i] = listArray[i+1];
  numInList--;                  // Decrement list size
  return it;
}

public void setFirst() { curr = 0; } // Set to first
public void prev()  { curr--; }  // Move curr to prev
public void next()  { curr++; }  // Move curr to next
public int length() { return numInList; }
public void setPos(int pos) { curr = pos; }
public boolean isEmpty() { return numInList == 0; }

public void setValue(Object it) { // Set current value
  Assert.notFalse(isInList(), "No current element");
  listArray[curr] = it;
}

public boolean isInList() // True if curr within list
  { return (curr >= 0) && (curr < numInList); }
} // Array-based list implementation
```

Link Class

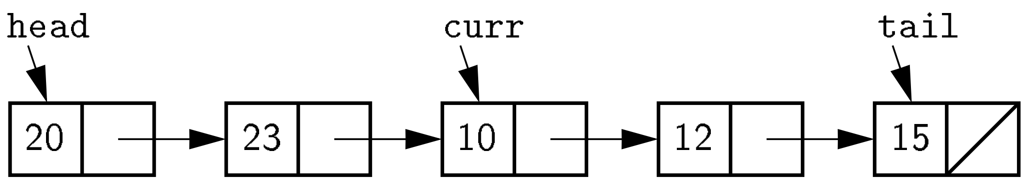
Dynamic allocation of new list elements.

```
class Link {                // A singly linked list node
  private Object element; // Object for this node
  private Link next;      // Pointer to next node
  Link(Object it, Link nextval)         // Constructor
    { element = it;  next = nextval; }
  Link(Link nextval) { next = nextval; } // Constructor
  Link next() { return next; }
  Link setNext(Link nextval) { return next = nextval; }
  Object element() { return element; }
  Object setElement(Object it) { return element = it; }
}
```

Linked List Position

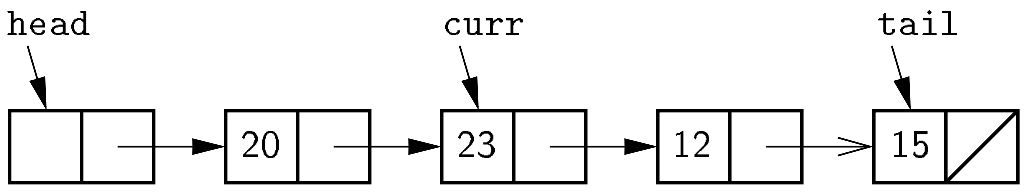


(a)

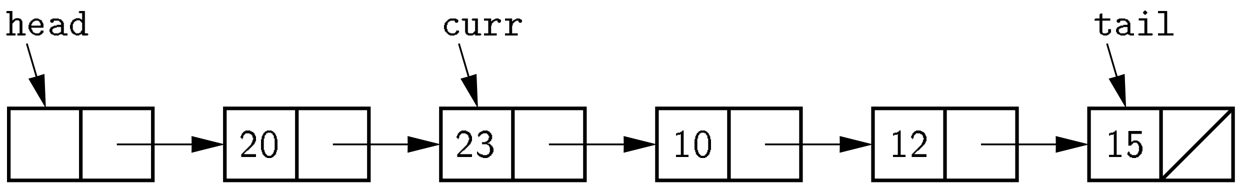


(b)

[Naive approach: Point to current node. Current is 12. Want to insert node with 10. No access available to node with 23. How can we do the insert?]



(a)



(b)

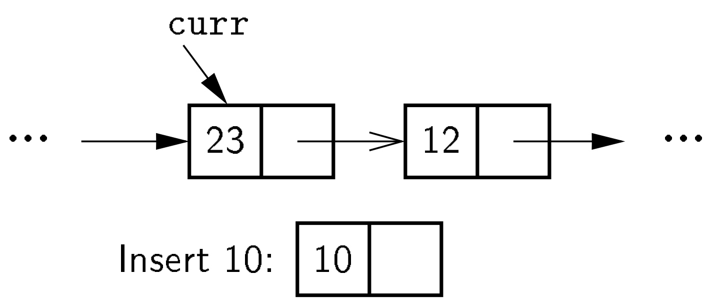
[Alt implementation: Point to node preceding actual current node. Now we can do the insert. Also note use of header node.]

Linked List Implementation

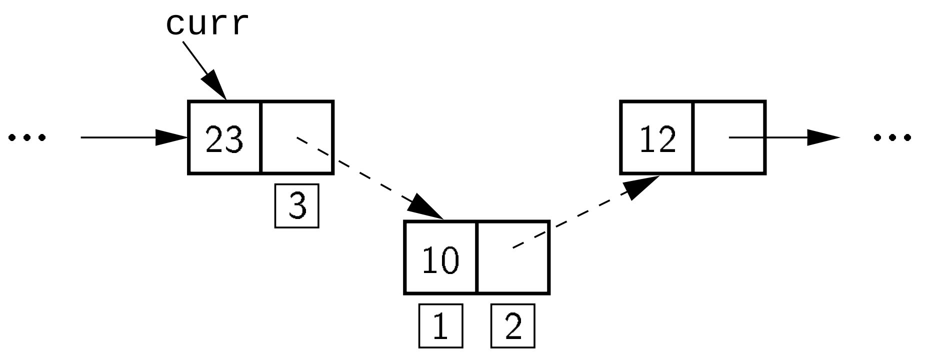
```
public class LList implements List { // Linked list
private Link head;    // Pointer to list header
private Link tail;    // Pointer to last Object in list
protected Link curr;  // Position of current Object

LList(int sz) { setup(); }     // Constructor
LList() { setup(); }           // Constructor
private void setup()           // allocates leaf node
  { tail = head = curr = new Link(null); }

public void setFirst() { curr = head; }
public void next()
  { if (curr != null) curr = curr.next(); }

public void prev() {       // Move to previous position
  Link temp = head;
  if ((curr == null) || (curr == head)) // No prev
    { curr = null;  return; }           //  so return
  while ((temp != null) && (temp.next() != curr))
        temp = temp.next();
  curr = temp;
}

public Object currValue() { // Return current Object
  if (!isInList() || this.isEmpty() ) return null;
  return curr.next().element();
}

public boolean isEmpty()    // True if list is empty
  { return head.next() == null; }
} // Linked list class
```

Linked List Insertion

```
// Insert Object at current position
public void insert(Object it) {
  Assert.notNull(curr, "No current element");
  curr.setNext(new Link(it, curr.next()));
  if (tail == curr)           // Appended new Object
    tail = curr.next();
}
```



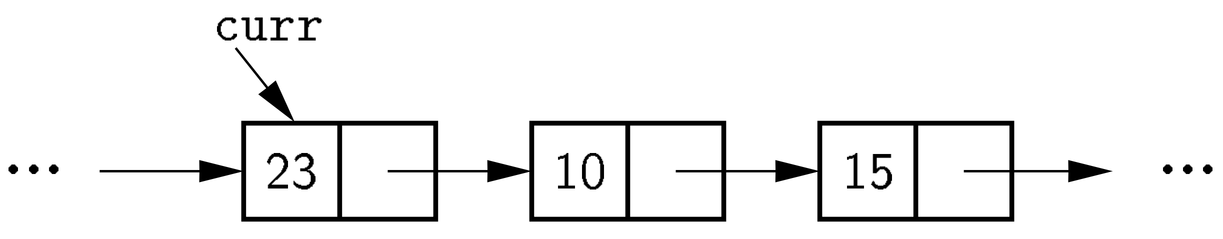
(a)



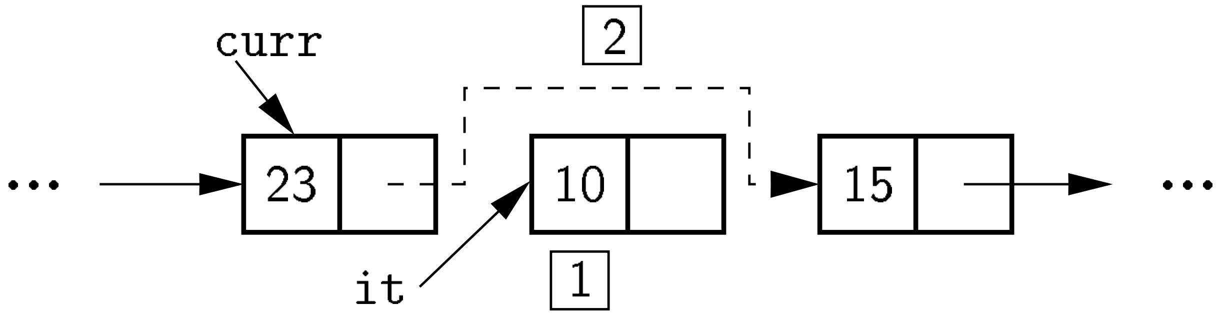
(b)

Linked List Remove

```
public Object remove() { // Remove/return curr Object
  if (!isInList() || this.isEmpty() ) return null;
  Object it = curr.next().element(); // Remember value
  if (tail == curr.next()) tail = curr; // Set tail
  curr.setNext(curr.next().next());  // Cut from list
  return it;                         // Return value
}
```



(a)



(b)

Freelists

System `new` and garbage collection are slow.

```
class Link {  // Singly linked list node with freelist
  private Object element; // Object for this Link
  private Link next;      // Pointer to next Link
  Link(Object it, Link nextval)
  { element = it;  next = nextval; }
  Link(Link nextval) { next = nextval; }
  Link next() { return next; }
  Link setNext(Link nextval) { return next = nextval; }
  Object element() { return element; }
  Object setElement(Object it) { return element = it; }

  // Extensions to support freelists
  static Link freelist = null; // Freelist for class

  static Link get(Object it, Link nextval) {
    if (freelist == null)//free list empty: allocate
      return new Link(it, nextval);
    Link temp = freelist; //take from the freelist
    freelist = freelist.next();
    temp.setElement(it);
    temp.setNext(nextval);
    return temp;
  }
  void release() {
    // add current node to freelist
    element = null;  next = freelist;  freelist = this;
  }
}
```

Comparison of List Implementations

Array-Based Lists: **[Average and worst cases]**

- Insertion and deletion are $\Theta(n)$.
- Array must be allocated in advance.
- No overhead if all array positions are full.

Linked Lists:

- Insertion and deletion $\Theta(1)$;
  prev and direct access are $\Theta(n)$.
- Space grows with number of elements.
- Every element requires overhead.

Space "break-even" point:

$$DE = n(P + E); \quad n = \frac{DE}{P + E}$$

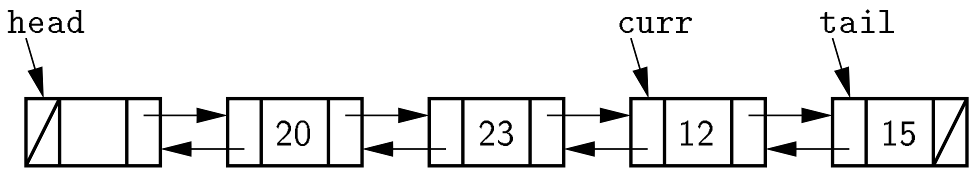
n: elements currently in list
E: Space for data value
P: Space for pointer
D: Number of elements in array (fixed in the implementation)

[arrays more efficient when full, linked lists more efficient with few elements]

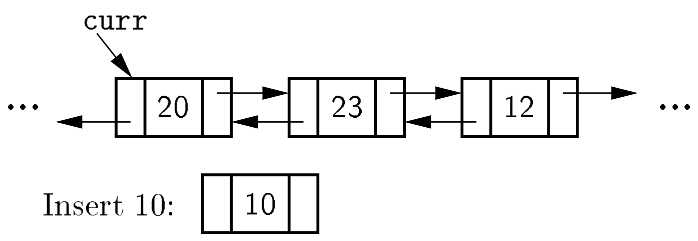
Doubly Linked Lists

Simplify insertion and deletion: Add a `prev` pointer.

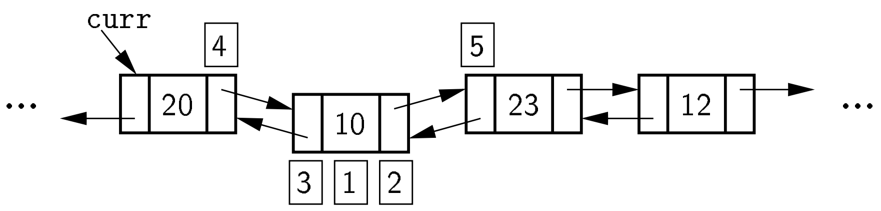
```
class DLink {               // A doubly-linked list node
  private Object element; // Object for this node
  private DLink next;     // Pointer to next node
  private DLink prev;     // Pointer to previous node
  DLink(Object it, DLink n, DLink p)
  { element = it;  next = n; prev = p; }
  DLink(DLink n, DLink p) { next = n;  prev = p; }
  DLink next() { return next; }
  DLink setNext(DLink nextval) { return next=nextval; }
  DLink prev() { return prev; }
  DLink setPrev(DLink prevval) { return prev=prevval; }
  Object element() { return element; }
  Object setElement(Object it) { return element = it; }
}
```



Doubly Linked List Operations



(a)



(b)

```
// Insert Object at current position
public void insert(Object it) {
  Assert.notNull(curr, "No current element");
  curr.setNext(new DLink(it, curr.next(), curr));
  if (curr.next().next() != null)
    curr.next().next().setPrev(curr.next());
  if (tail == curr)                // Appended new Object
    tail = curr.next();
}

public Object remove() { // Remove/return curr Object
  Assert.notFalse(isInList(), "No current element");
  Object it = curr.next().element(); // Remember Object
  if (curr.next().next() != null)
    curr.next().next().setPrev(curr);
  else tail = curr;  // Removed last Object: set tail
  curr.setNext(curr.next().next()); // Remove from list
  return it;                       // Return value removed
}
```

Circularly Linked Lists

- Convenient if there is no last nor first element (there is no total order among elements)
- The "last" element points to the "first", and the first to the last
- tail pointer non longer needed
- Potential danger: infinite loops in list processing
- but head pointer can be used as a marker

Stacks

LIFO: Last In, First Out

Restricted form of list: Insert and remove only at front of list.

Notation:

- Insert: PUSH
- Remove: POP
- The accessible element is called TOP.

Array-Based Stack

Define `top` as first free position.

```
class AStack implements Stack{ // Array based stack class
  private static final int defaultSize = 10;
  private int size;            // Maximum size of stack
  private int top;             // Index for top Object
  private Object [] listarray; // Array holding stack
  AStack() { setup(defaultSize); }
  AStack(int sz) { setup(sz); }

  public void setup(int sz)
  { size = sz;  top = 0; listarray = new Object[sz]; }

  public void clear() { top = 0; } // Clear all Objects

  public void push(Object it) // Push onto stack
  { Assert.notFalse(top < size, "Stack overflow");
    listarray[top++] = it; }

  public Object pop()         // Pop Object from top
  { Assert.notFalse(!isEmpty(), "Empty stack");
    return listarray[--top]; }

  public Object topValue()    // Return top Object
  { Assert.notFalse(!isEmpty(), "Empty stack");
    return listarray[top-1]; }

  public boolean isEmpty() { return top == 0; }
};
```

Linked Stack

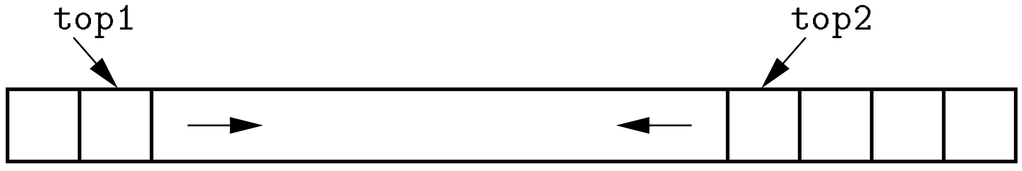
```
public class LStack implements Stack {
                   // Linked stack class
private Link top;       // Pointer to list header

public LStack() { setup(); }        // Constructor
public LStack(int sz) { setup(); }  // Constructor

private void setup()      // Initialize stack
{ top = null; }           // Create header node

public void clear() { top = null; } // Clear stack

public void push(Object it) // Push Object onto stack
{ top = new Link(it, top); }

public Object pop() {    // Pop Object from top
  Assert.notFalse(!isEmpty(), "Empty stack");
  Object it = top.element();
  top = top.next();
  return it;
}

public Object topValue() // Get value of top Object
{ Assert.notFalse(!isEmpty(), "No top value");
  return top.element(); }

public boolean isEmpty() // True if stack is empty
{ return top == null; }
} // Linked stack class
```

Array-based vs linked stacks

- Time: all operations take constant time for both
- Space: linked has overhead but is flexible; array has no overhead but wastes space when not full

Implementation of multiple stacks

- two stacks at opposite ends of an array growing in opposite directions
- works well if their space requirements are inversely correlated



Queues

FIFO: First In, First Out

Restricted form of list:
Insert at one end, remove from other.

Notation:

- Insert: Enqueue
- Delete: Dequeue
- First element: FRONT
- Last element: REAR

Array Queue Implementations

Constraint: all elements

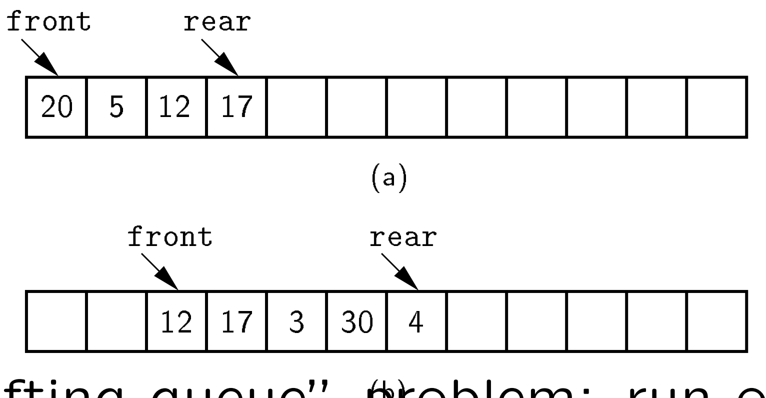
1. in consecutive positions
2. in the initial (final) portion of the array

If both (1) and (2) hold: rear element in pos 0, dequeue costs $\Theta(1)$, enqueue costs $\Theta(n)$

Similarly if in final portion of the array and/or in reverse order

If only (1) holds (2 is released)

- both front and rear move to the "right" (i.e., increase)
- both enqueue and dequeue cost $\Theta(1)$



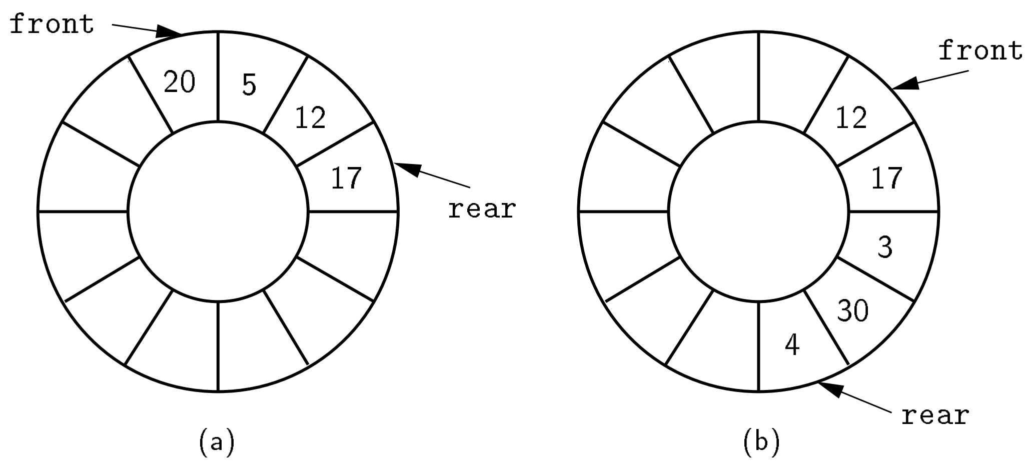
"Drifting queue" problem: run out of space when at the highest posistions

Solution: pretend the array is circular, implemented by the modulus operator, e.g., front = (front + 1) % size

Array Q Impl (cont)

A more serious problem: empty queue indistinguishable from full queue

[Application of Pigeonhole Principle: Given a fixed (arbitrary) position for front, there are $n+1$ states (0 through n elements in queue) and only n positions for rear. One must distinguish between two of the states.]



2 solutions to this problem

1. store # elements separately from the queue
2. use a $n+1$ elements array for holding a queue with n elements an most

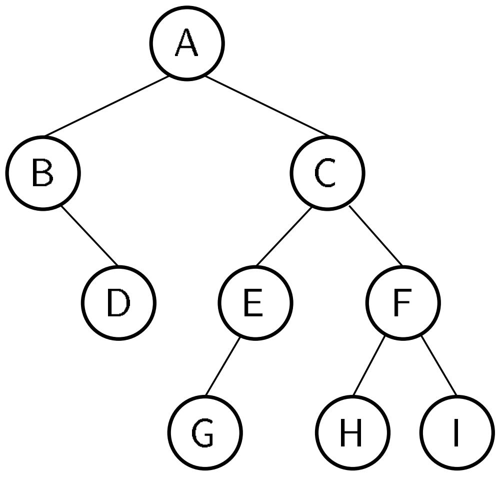
Both solutions require one additional item of information

Linked Queue: modified linked list.

[Operations are $\Theta(1)$]

Binary Trees

A **binary tree** is made up of a finite set of nodes that is either empty (then it is an **empty** tree) or consists of a node called the **root** connected to two binary trees, called the left and right **subtrees**, which are disjoint from each other and from the root.



[A has depth 0. B and C form level 1. The tree has height 4. Height = max depth + 1.]

Notation

(left/right) child of a node: root node of the (left/right) subtree

if there is no left (right) subtree we say that left/(right) subtree is **empty**

edge: connection between a node and its child (drawn as a line)

parent of a node n: the node of which n is a child

path from n_1 to n_k: a sequence n_1 n_2 ... n_k, $k >= 1$, such that, for all $1 <= i < k$, n_i is parent of n_{i+1}

length of a path n_1 n_2 ... n_k is $k - 1$ ($\Rightarrow$ length of path n_1 is 0)

if there is a path from node a to node d then

- a is **ancestor** of d
- d is **descendant** of a

Notation (Cont.)

hence

- all nodes of a tree (except the root) are descendant of the root
- the root is ancestor of all the other nodes of the tree (except itself)

depth of a node: length of a path from the root ($\Rightarrow$ the root has depth 0)

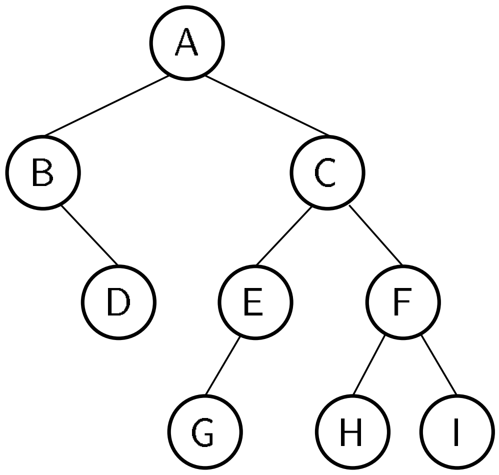
height of a tree: 1 + depth of the deepest node (which is a leaf)

level d of a tree: the set of all nodes of depth d ($\Rightarrow$ root is the only node of level 0)

leaf node: has two empty children

internal node (non-leaf): has at least one non-empty child

Examples



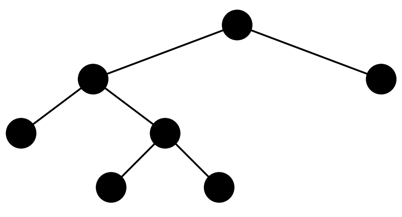
- A: root
- B, C: A's children
- B, D: A's subtree
- D, E, F: level 2
- B has only right child (subtree)
- path of length 3 from A to G
- A, B, C, E, F internal nodes
- D, G, H, I leaves
- depth of G is 3, height of tree is 4

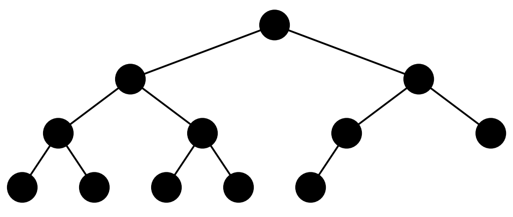
Full and Complete Binary Trees

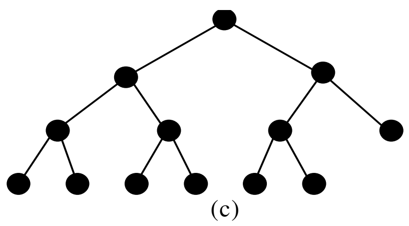
Full binary tree: each node either is a leaf or is an internal node with exactly two non-empty children.

Complete binary tree: If the height of the tree is d, then all levels except possibly level $d-1$ are completely full. The bottom level has nodes filled in from the left side.

(a) full but not complete
(b) complete but not full
(c) full and complete


(a)


(b)


(c)

[NB these terms can be hard to distinguish

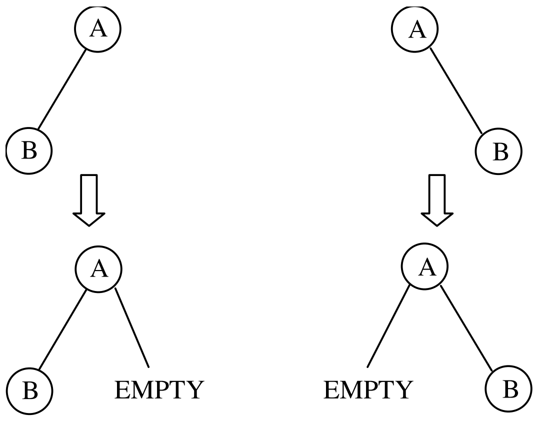
Question: how many nodes in a complete binary tree?

A complete binary tree is "balanced", i.e., has minimal height given number of nodes

A complete binary tree is full or almost full or "almost full"

(at most one node with one son)]

Making missing children explicit



for a (non-)empty subtree we say the node has a (non-)NULL pointer

Full Binary Tree Theorem

Theorem: The number of leaves in a non-empty full binary tree is one more than the number of internal nodes.

[Relevant since it helps us calculate space requirements.]

Proof (by Mathematical Induction):

- **Base Case**: A full binary tree with 0 internal node has 1 leaf node.
- **Induction Hypothesis**: Assume any full binary tree **T** containing $n-1$ internal nodes has n leaves.
- **Induction Step**: Given a full tree **T** with $n-1$ internal nodes ($\Rightarrow n$ leaves), add two leaf nodes as children of one of its leaves $\Rightarrow$ obtain a tree **T'** having n internal nodes and $n+1$ leaves.

Full Binary Tree Theorem Corollary

Theorem: The number of empty subtrees in a non-empty binary tree is one more than the number of nodes in the tree.

Proof: Replace all empty subtrees with a leaf node. This is a full binary tree, having #leaves = #empty subtrees of original tree.

alternative **Proof**:

- by definition, every node has 2 children, whether empty or not
- hence a tree with n nodes has $2n$ children
- every node (except the root) has 1 parent
  - $\Rightarrow$ there are $n-1$ parent nodes (some coincide)
  - $\Rightarrow$ there are $n-1$ non-empty children
- hence #(empty children) = #(total children) - #(non-empty children) = $2n-(n-1)=n+1$.

Binary Tree Node ADT

```
interface BinNode { // ADT for binary tree nodes
  // Return and set the element value
  public Object element();
  public Object setElement(Object v);

  // Return and set the left child
  public BinNode left();
  public BinNode setLeft(BinNode p);

  // Return and set the right child
  public BinNode right();
  public BinNode setRight(BinNode p);

  // Return true if this is a leaf node
  public boolean isLeaf();
} // interface BinNode
```

Traversals

Any process for visiting the nodes in some order is called a **traversal**.

Any traversal that lists every node in the tree exactly once is called an **enumeration** of the tree's nodes.

Preorder traversal: Visit each node *before* visiting its children.

Postorder traversal: Visit each node *after* visiting its children.

Inorder traversal: Visit the left subtree, then the node, then the right subtree.

NB: an empty node (tree) represented by Java's **null** (object) value

```
void preorder(BinNode rt) // rt is root of subtree
{
  if (rt == null) return; // Empty subtree
  visit(rt);
  preorder(rt.left());
  preorder(rt.right());
}
```

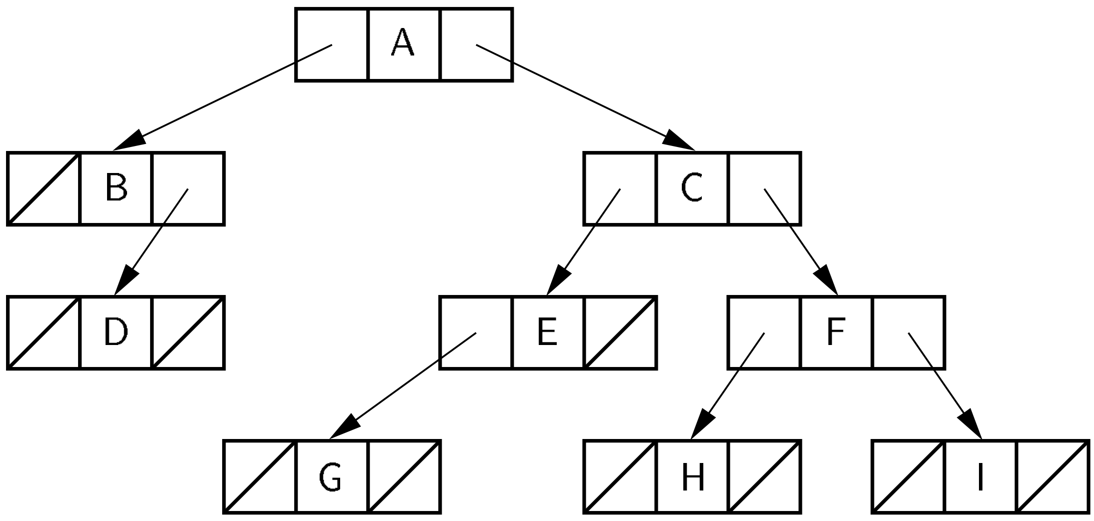
Traversals (cont.)

This is a $left-to-right$ preorder: first visit $left$ subtree, then the $right$ one.

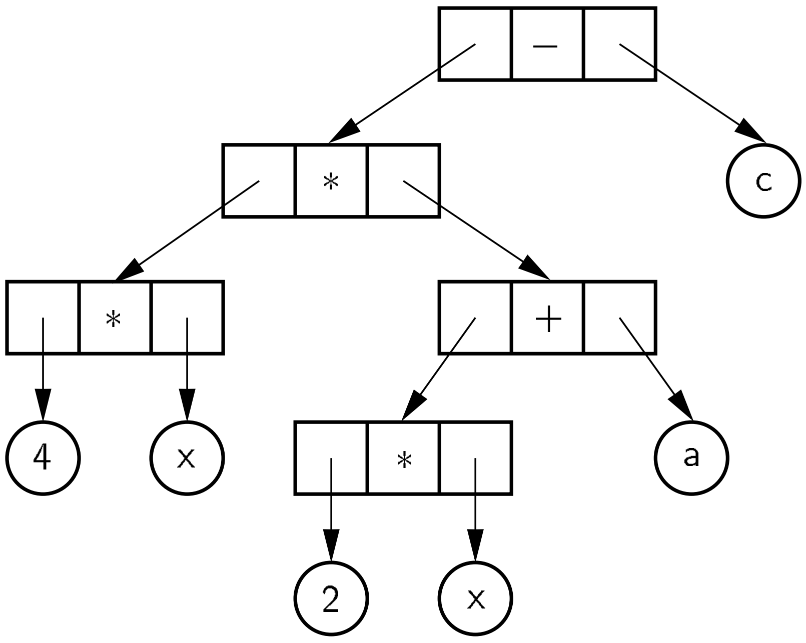
Get a $right-to-left$ preorder by switching last two lines

To get inorder or postorder, just rearrange the last three lines.

Binary Tree Implementation



[Leaves are the same as internal nodes. Lots of wasted space.]



[Example of expression tree: $(4x * (2x + a)) - c$. Leaves are different from internal nodes.]

Two implementations of BinNode

```
class LeafNode implements BinNode { // Leaf node
  private String var;               // Operand value

  public LeafNode(String val) { var = val; }
  public Object element() { return var; }
  public Object setElement(Object v)
    { return var = (String)v; }
  public BinNode left() { return null; }
  public BinNode setLeft(BinNode p) { return null; }
  public BinNode right() { return null; }
  public BinNode setRight(BinNode p) { return null; }
  public boolean isLeaf() { return true; }
} // class LeafNode

class IntlNode implements BinNode { // Internal node
  private BinNode left;             // Left child
  private BinNode right;            // Right child
  private Character opx;            // Operator value

  public IntlNode(Character op, BinNode l, BinNode r)
    { opx = op; left = l; right = r; } // Constructor
  public Object element() { return opx; }
  public Object setElement(Object v)
    { return opx = (Character)v; }
  public BinNode left() { return left; }
  public BinNode setLeft(BinNode p) {return left = p;}
  public BinNode right() { return right; }
  public BinNode setRight(BinNode p)
    { return right = p; }
  public boolean isLeaf() { return false; }
} // class IntlNode
```

Two implementations (cont)

```
static void traverse(BinNode rt) { // Preorder
  if (rt == null) return;          // Nothing to visit
  if (rt.isLeaf())                 // Do leaf node
    System.out.println("Leaf: " + rt.element());
  else {                           // Do internal node
    System.out.println("Internal: " + rt.element());
    traverse(rt.left());
    traverse(rt.right());
  }
}
```

A note on polymorphism and dynamic binding

The member function **isLeaf()** allows one to distinguish the “type” of a node

- leaf
- internal

without need of knowing its subclass

This is determined dynamically by the JRE (Java Runtime Environment)

Space Overhead

From Full Binary Tree Theorem:

Half of pointers are `NULL`.

If leaves only store information, then overhead depends on whether tree is full.

All nodes the same, with two pointers to children:

Total space required is $(2p + d)n$.

Overhead: $2pn$.

If $p = d$, this means $2p/(2p + d) = 2/3$ overhead.

[The following is for full binary trees:]

Eliminate pointers from leaf nodes:

$$\frac{\frac{n}{2}(2p)}{\frac{n}{2}(2p) + dn} = \frac{p}{p + d}$$

[Half the nodes have 2 pointers, which is overhead.]

This is 1/2 if $p = d$.

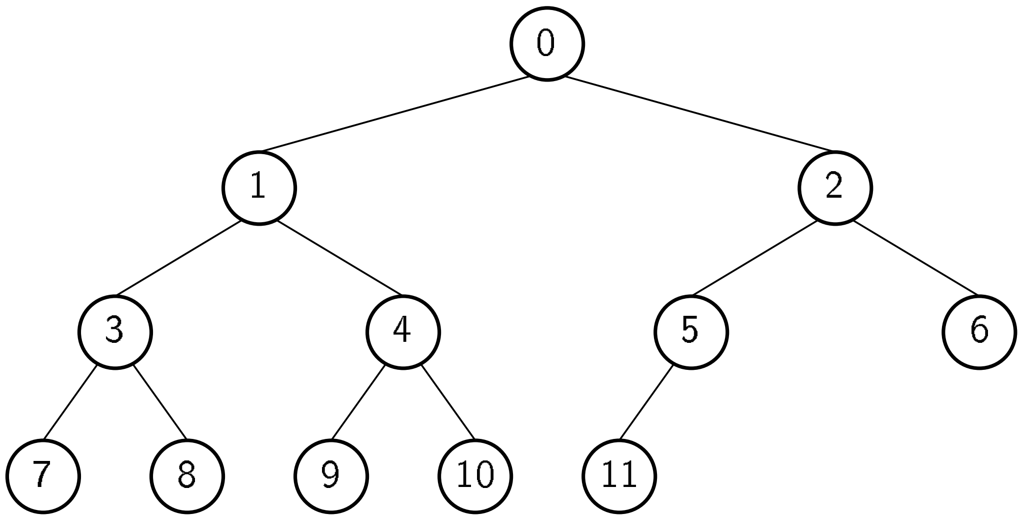
$2p/(2p + d)$ if data only at leaves $\Rightarrow$ 2/3 overhead.

Some method is needed to distinguish leaves from internal nodes. **[This adds overhead.]**

Array Implementation

[This is a good example of logical representation vs. physical implementation.]

For complete binary trees.



(a)

Node	0	1	2	3	4	5	6	7	8	9	10	11

- Parent(r) = **[**$(r-1)/2$ **if** $r \neq 0$ **and** $r < n$.**]**
- Leftchild(r) = **[**$2r+1$ **if** $2r+1 < n$.**]**
- Rightchild(r) = **[**$2r+2$ **if** $2r+2 < n$.**]**
- Leftsibling(r) = **[**$r-1$ **if** r **is even,** $r > 0$ **and** $r < n$.**]**
- Rightsibling(r) = **[**$r+1$ **if** r **is odd,** $r+1 < n$.**]**

  [Since the complete binary tree is so limited in its shape, (only one shape for tree of n nodes), it is reasonable to expect that space efficiency can be achieved.

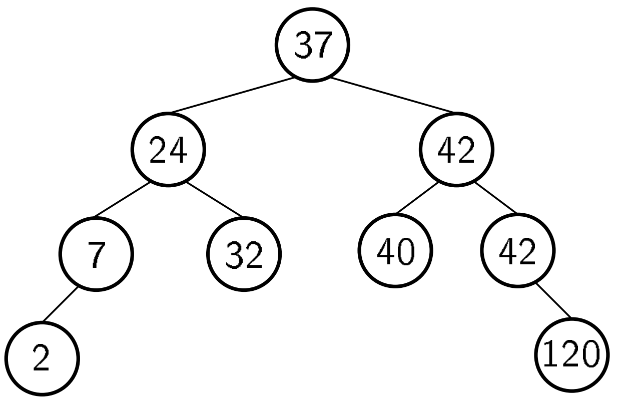
  NB: left sons' indices are always odd, right ones' even, a node with index i is leaf iff $i > n.of.nodes/2$ (Full Binary Tree Theorem)]

Binary Search Trees

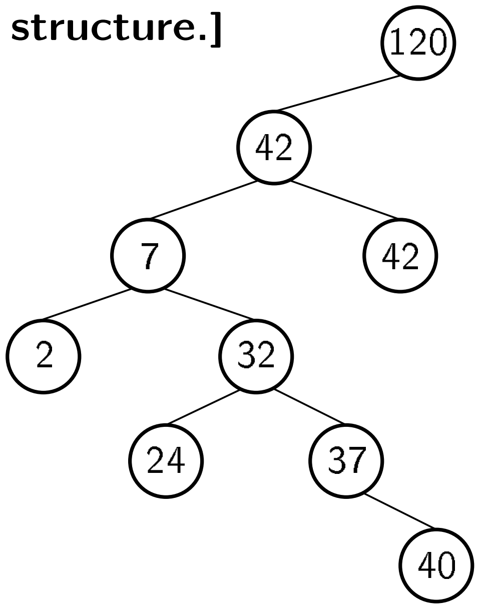
Binary Search Tree (BST) Property

All elements stored in the left subtree of a node whose value is K have values less than K. All elements stored in the right subtree of a node whose value is K have values greater than or equal to K.

[Problem with lists: either insert/delete or search must be $\Theta(n)$ time. How can we make both update and search efficient? Answer: Use a new data structure.]



(a)



(b)

BinNode Class

```
interface BinNode { // ADT for binary tree nodes
  // Return and set the element value
  public Object element();
  public Object setElement(Object v);

  // Return and set the left child
  public BinNode left();
  public BinNode setLeft(BinNode p);

  // Return and set the right child
  public BinNode right();
  public BinNode setRight(BinNode p);

  // Return true if this is a leaf node
  public boolean isLeaf();
} // interface BinNode
```

We assume that the datum in the nodes implements interface Elem with a method key used for comparisons (in searching and sorting algorithms)

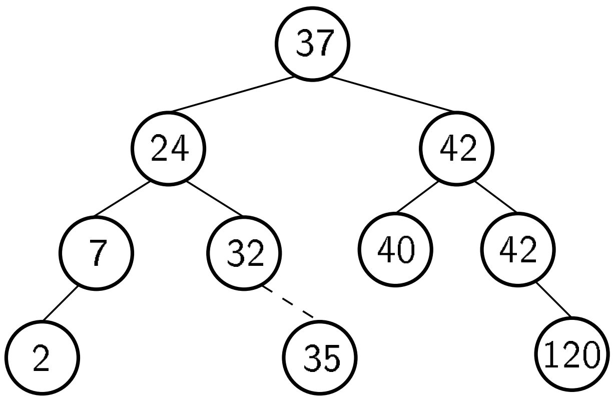
```
interface Elem {
  public abstract int key();
} // interface Elem
```

BST Search

```
public class BST { // Binary Search Tree implementation
  private BinNode root; // The root of the tree

  public BST() { root = null; } // Initialize root
  public void clear() { root = null; }
  public void insert(Elem val)
    { root = inserthelp(root, val); }
  public void remove(int key)
    { root = removehelp(root, key); }
  public Elem find(int key)
    { return findhelp(root, key); }
  public boolean isEmpty() { return root == null; }

  public void print() {
    if (root == null)
      System.out.println("The BST is empty.");
    else {
      printhelp(root, 0);
      System.out.println();
    }
  }

private Elem findhelp(BinNode rt, int key) {
  if (rt == null) return null;
  Elem it = (Elem)rt.element();
  if (it.key() > key)  return findhelp(rt.left(), key);
  else if (it.key() == key)  return it;
  else  return findhelp(rt.right(), key);
}
```

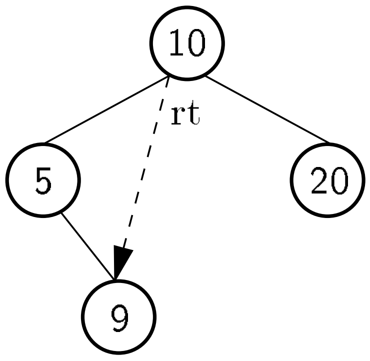
BST Insert

```
private BinNode inserthelp(BinNode rt, Elem val) {
  if (rt == null) return new BinNode(val);
  Elem it = (Elem) rt.element();
  if (it.key() > val.key())
    rt.setLeft(inserthelp(rt.left(), val));
  else
    rt.setRight(inserthelp(rt.right(), val));
  return rt;
}
```



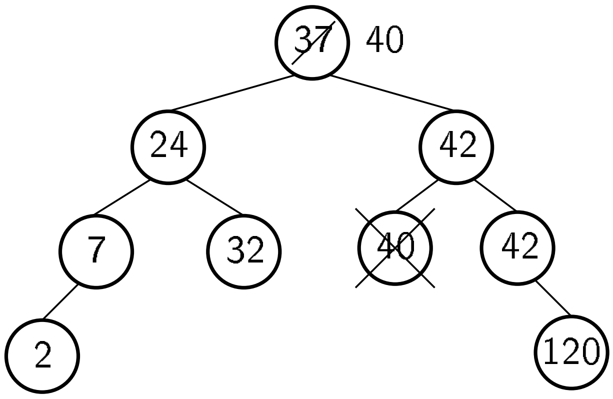
Remove Minimum Value

```
private BinNode deletemin(BinNode rt) {
  if (rt.left() == null)
    return rt.right();
  else {
    rt.setLeft(deletemin(rt.left()));
    return rt;
  }
}

private Elem getmin(BinNode rt) {
  if (rt.left() == null)
    return (Elem)rt.element();
  else return getmin(rt.left());
}
```



BST Remove

```
private BinNode removehelp(BinNode rt, int key) {
  if (rt == null) return null;
  Elem it = (Elem) rt.element();
  if (key < it.key())
    rt.setLeft(removehelp(rt.left(), key));
  else if (key > it.key())
    rt.setRight(removehelp(rt.right(), key));
  else {
    if (rt.left() == null)
      rt = rt.right();
    else if (rt.right() == null)
      rt = rt.left();
    else {
      Elem temp = getmin(rt.right());
      rt.setElement(temp);
      rt.setRight(deletemin(rt.right()));
    }
  }
  return rt;
}
```



Cost of BST Operations

Find: the depth of the node being found

Insert: the depth of the node being inserted

Remove: the depth of the node being removed, if it has < 2 children, otherwise depth of node with smallest value in its right subtree

Best case: balanced (complete tree): $\Theta(\log n)$

Worst case (linear tree): $\Theta(n)$

That's why it is important to have a balanced (complete) BST

Cost of *constructing* a BST by means of a series of insertions

- if elements inserted in in order of increasing value $\sum_{i=1}^{n} i = \Theta(n^2)$
- if inserted in "random" order almost good enough for balancing the tree, insertion cost is in average $\Theta(\log n)$, for a total $\Theta(n \log n)$

Heaps

Heap: Complete binary tree with the **Heap Property**:

- Min-heap: all values less than child values.
- Max-heap: all values greater than child values.

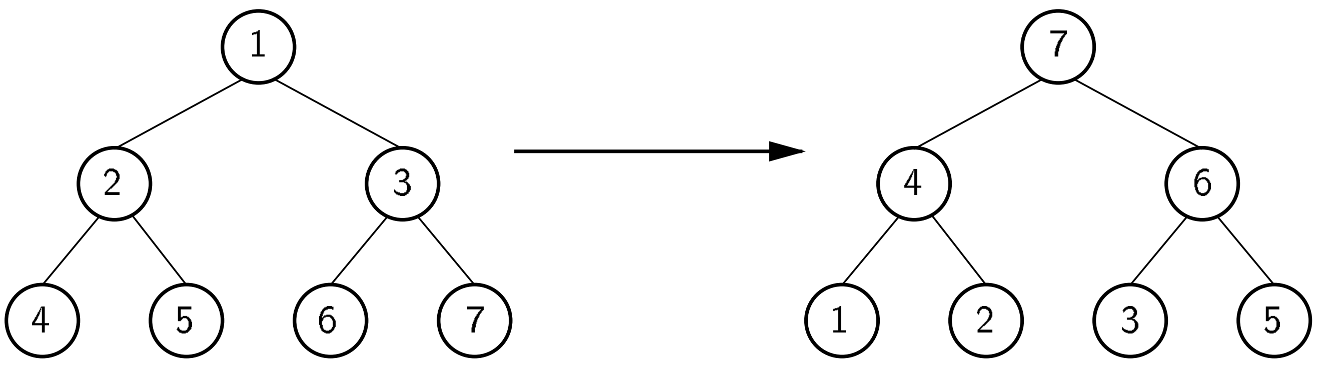
The values in a heap are **partially ordered**.

Heap representation: normally the array based complete binary tree representation.

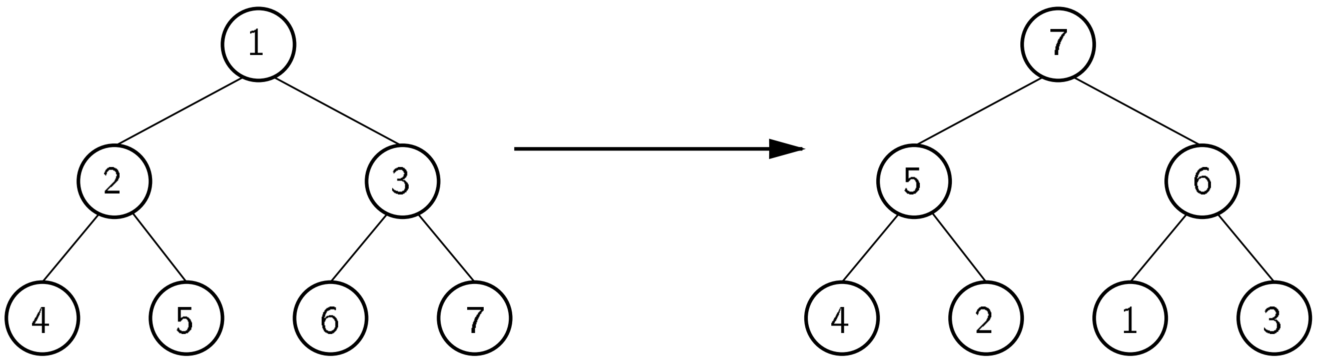
Building the Heap

[Max Heap

NB: for a given set of values, the heap is not unique]



(a)



(b)

(a) requires exchanges (4-2), (4-1), (2-1), (5-2), (5-4), (6-3), (6-5), (7-5), (7-6).

(b) requires exchanges (5-2), (7-3), (7-1), (6-1).

[How to get a good number of exchanges? By induction. Heapify the root's subtrees, then push the root to the correct level.]

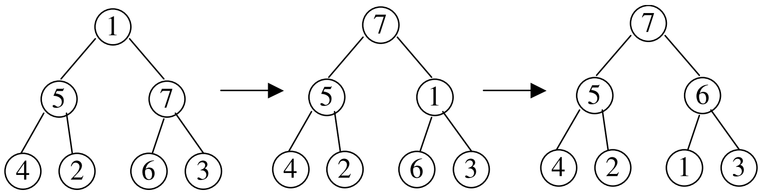
The siftdown procedure

To place a generic node in its correct position

Assume subtrees are Heaps

If root is not greater than both children, swap with greater child

Reapply on modified subtree



Shift it down by exchanging it with the greater of the two sons, until it becomes a leaf or it is greater than both sons.

Max Heap Implementation

```
public class MaxHeap {
private Elem[] Heap;  // Pointer to the heap array
private int size;     // Maximum size of the heap
private int n;        // Number of elements now in heap

public MaxHeap(Elem[] h, int num, int max)
{ Heap = h;  n = num;  size = max;  buildheap(); }

public int heapsize() // Return current size of heap
{ return n; }

public boolean isLeaf(int pos)  // TRUE if pos is leaf
{ return (pos >= n/2) && (pos < n); }

// Return position for left child of pos
public int leftchild(int pos) {
  Assert.notFalse(pos < n/2, "No left child");
  return 2*pos + 1;
}

// Return position for right child of pos
public int rightchild(int pos) {
  Assert.notFalse(pos < (n-1)/2, "No right child");
  return 2*pos + 2;
}

public int parent(int pos) { // Return pos for parent
  Assert.notFalse(pos > 0, "Position has no parent");
  return (pos-1)/2;
}
```

Siftdown

For fast heap construction:

- Work from high end of array to low end.
- Call `siftdown` for each item.
- Don't need to call `siftdown` on leaf nodes.

```
public void buildheap() // Heapify contents of Heap
  { for (int i=n/2-1; i>=0; i--) siftdown(i); }

private void siftdown(int pos) { // Put in place
  Assert.notFalse((pos >= 0) && (pos < n),
                  "Illegal heap position");
  while (!isLeaf(pos)) {
    int j = leftchild(pos);
    if ((j<(n-1)) && (Heap[j].key() < Heap[j+1].key()))
      j++; // j now index of child with greater value
    if (Heap[pos].key() >= Heap[j].key()) return;
    DSutil.swap(Heap, pos, j);
    pos = j;  // Move down
  }
}
```

Cost for heap construction

$$\sum_{i=1}^{\log n} (i-1)\frac{n}{2^i} = \Theta(n).$$

[$(i-1)$ is number of steps down, $n/2^i$ is number of nodes at that level.]

cfr. eq(2.7) p.28:

$$\sum_{i=1}^{n} \frac{i}{2^i} = 2 - \frac{n+2}{2^n}$$

notice that

$$\sum_{i=1}^{\log n}(i-1)\frac{n}{2^i} \leq n\sum_{i=1}^{n}\frac{i}{2^i}$$

Cost of removing root is $\Theta(\log n)$

Remove element too (root is a special case thereof)

Priority Queues

A priority queue stores objects, and on request releases the object with greatest value.

Example: Scheduling jobs in a multi-tasking operating system.

The **priority** of a job may change, requiring some reordering of the jobs.

Implementation: use a heap to store the priority queue.

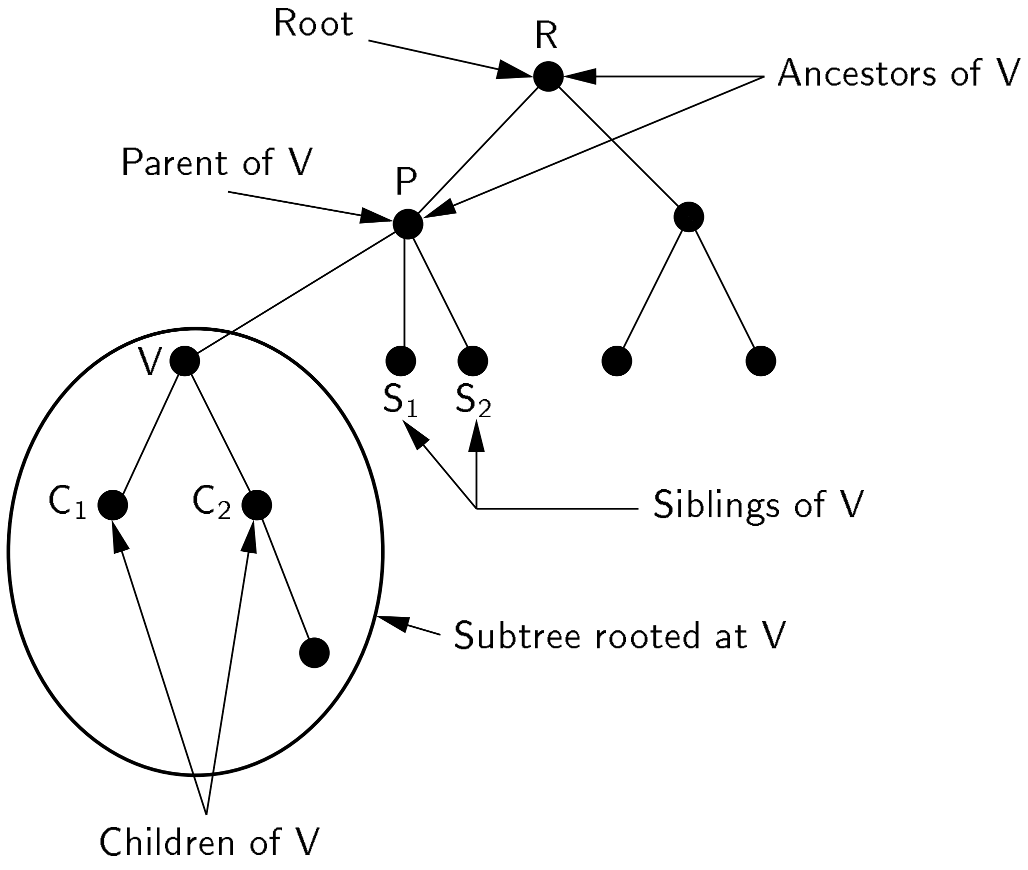
To support priority reordering, delete and re-insert. Need to know index for the object.

```
// Remove value at specified position
public Elem remove(int pos) {
  Assert.notFalse((pos >= 0) && (pos < n),
                  "Illegal heap position");
  DSutil.swap(Heap, pos, --n); // Swap with last value
  while (Heap[pos].key() > Heap[parent(pos)].key())
    DSutil.swap(Heap, pos, parent(pos)); // push up
  if (n != 0) siftdown(pos);             // push down
  return Heap[n];
}
```

General Trees

A **tree** T is a finite set of nodes such that it is empty or there is one designated node r called the root of T, and the remaining nodes in $(T - \{r\})$ are partitioned into $n \geq 0$ disjoint subsets T_1, T_2, ..., T_k, each of which is a tree.

[Note: disjoint because a node cannot have two parents.]



General Tree ADT

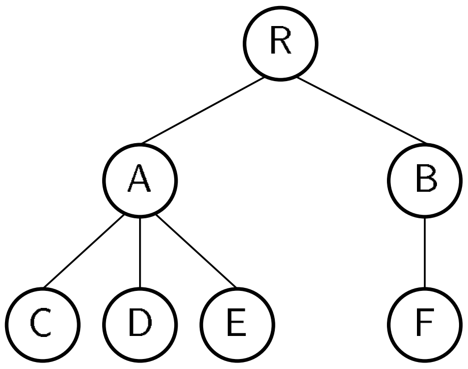
[There is no concept of “left” or “right” child. But, we can impose a concept of “first” (leftmost) and “next” (right).]

```
public interface GTNode {
  public Object value();
  public boolean isLeaf();
  public GTNode parent();
  public GTNode leftmost_child();
  public GTNode right_sibling();
  public void setValue(Object value);
  public void setParent(GTNode par);
  public void insert_first(GTNode n);
  public void insert_next(GTNode n);
  public void remove_first(); // remove first child
  public void remove_next();  // remove right sibling
}

public interface GenTree {
  public void clear();
  public GTNode root();
  public void newroot(Object value, GTNode first,
                                    GTNode sib);
}
```

General Tree Traversal

[preorder traversal]

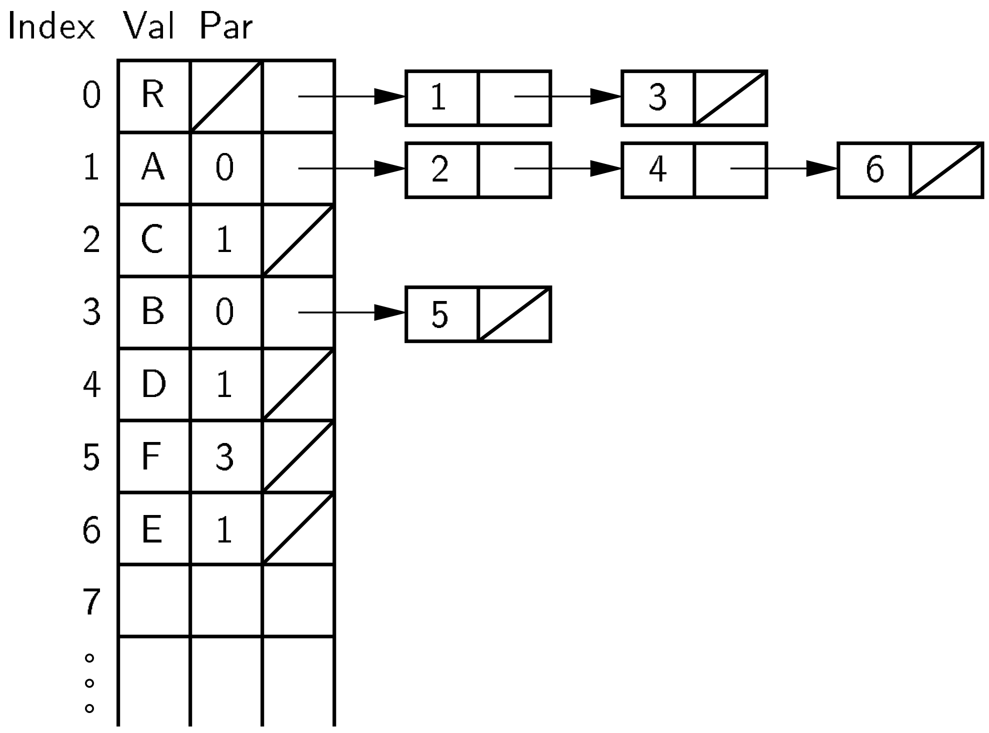
```
static void print(GTNode rt) { // Preorder traversal
  if (rt.isLeaf()) System.out.print("Leaf: ");
  else System.out.print("Internal: ");
  System.out.println(rt.value());
  GTNode temp = rt.leftmost_child();
  while (temp != null) {
    print(temp);
    temp = temp.right_sibling();
  }
}
```



[RACDEBF]

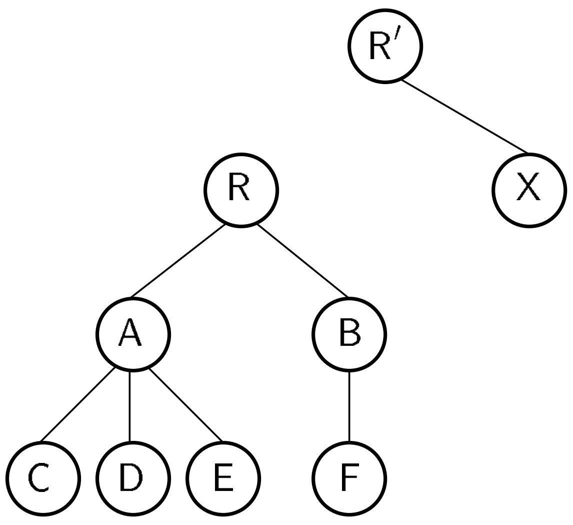
General Tree Implementations

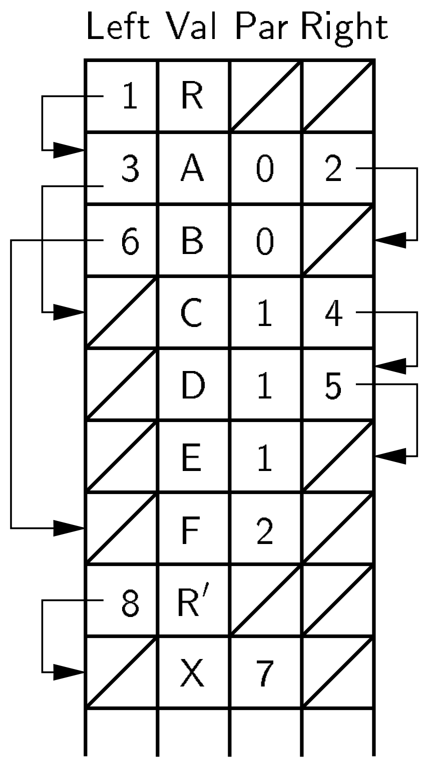
Lists of Children



[Hard to find right sibling.]

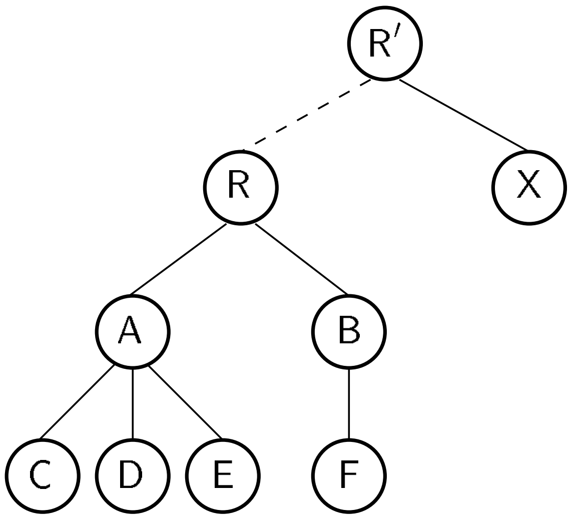
Leftmost Child/Right Sibling

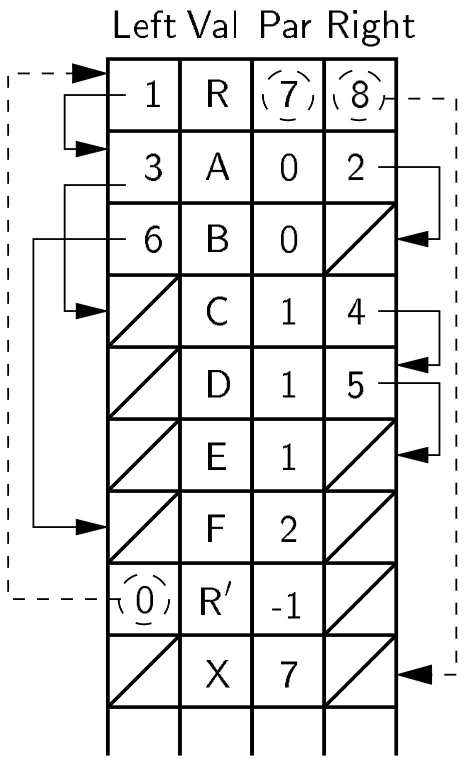




Left	Val	Par	Right
1	R		
3	A	0	2
6	B	0	
	C	1	4
	D	1	5
	E	1	
	F	2	
8	R′		
	X	7	

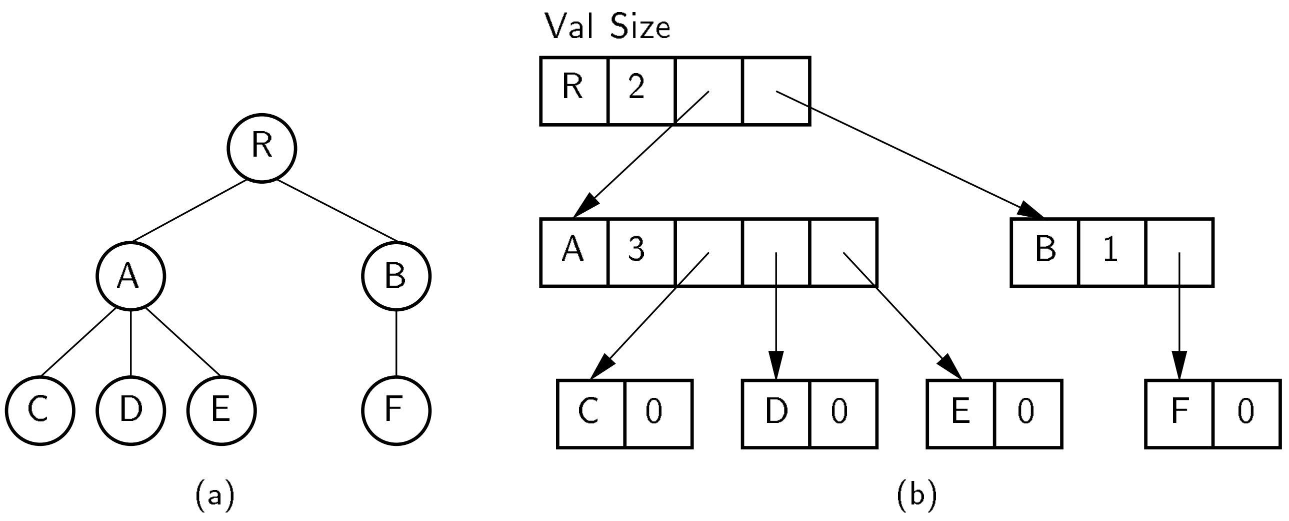
[Note: Two trees share same array.]



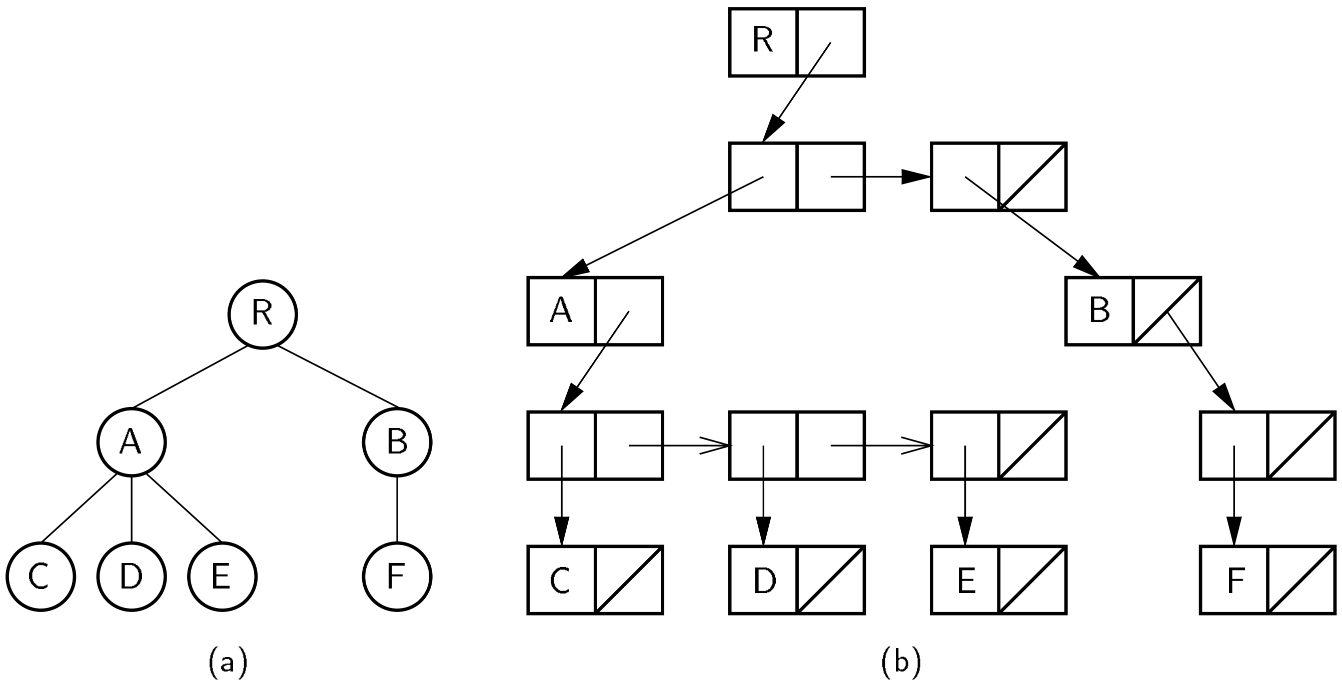


Left	Val	Par	Right
1	R	7	8
3	A	0	2
6	B	0	
	C	1	4
	D	1	5
	E	1	
	F	2	
0	R′	-1	
	X	7	

Linked Implementations



[Allocate child pointer space when node is created.]



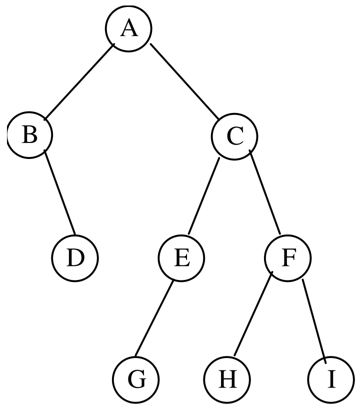
Sequential Implementations

List node values in the order they would be visited by a preorder traversal.

Saves space, but allows only sequential access.

Need to retain tree structure for reconstruction.

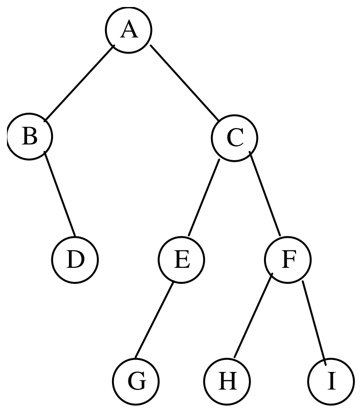
For binary trees: Use symbol to mark `NULL` links.



$$AB/D//CEG///FH//I//$$

Sequential Implementations (cont.)

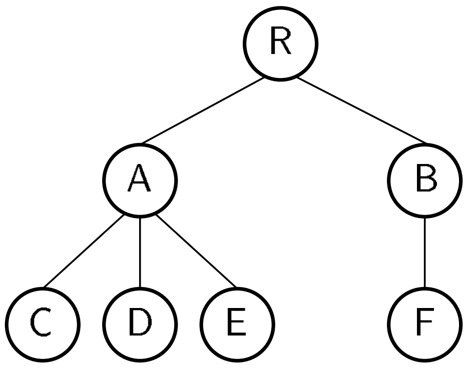
Full binary trees: Mark leaf or internal.



[Need NULL mark since this tree is not full.]

$$A'B'/DC'E'G/F'HI$$

General trees: Mark end of each subtree.



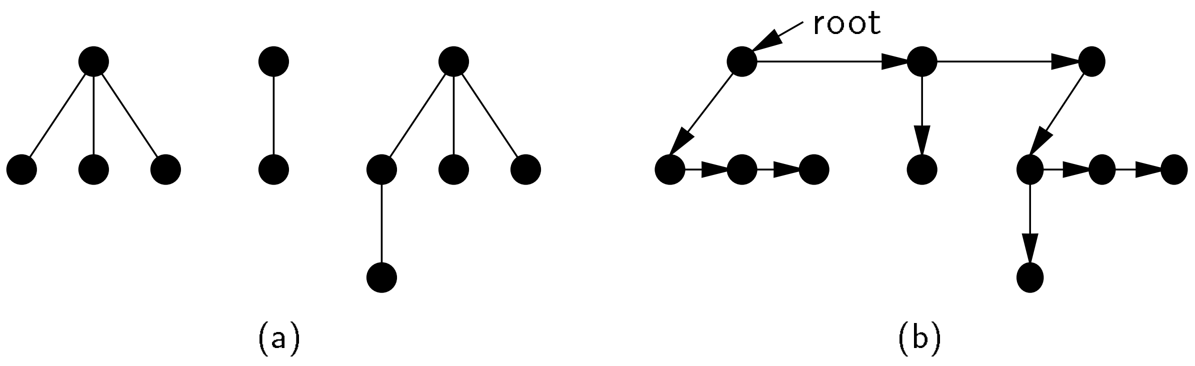
$$RAC)D)E))BF)))$$

Convert to Binary Tree

Left Child/Right Sibling representation essentially stores a binary tree.

Use this process to convert any general tree to a binary tree.

A **forest** is a collection of one or more general trees.



[Dynamic implementation of "Left child/right sibling."]

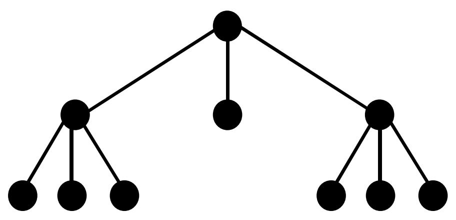
K-ary Trees

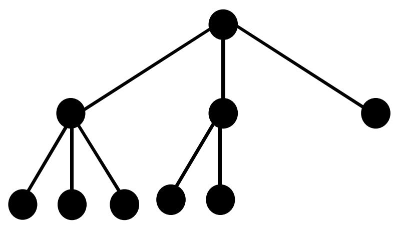
Every node has a fixed maximum number of children

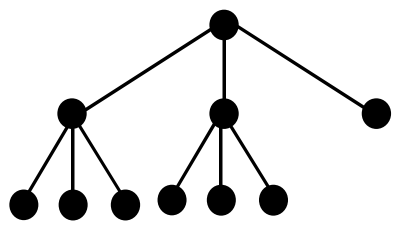
fixed # children ⇒ easy to implement, also in array

K high ⇒ potentially many empty subtrees ⇒ different implementation for leaves becomes convenient

Full and complete K-ary trees similar to binary trees


full, not complete


complete, not full


full and complete

Theorems on # empty subtrees and on relation between # internal nodes and # leaves similar to binary trees

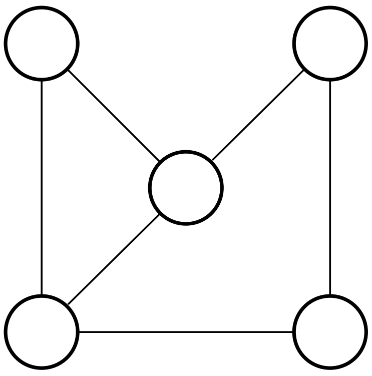
Graphs

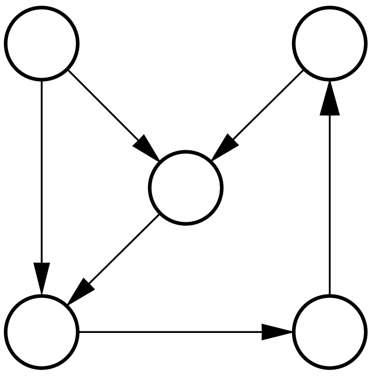
graph G = (V, E): a set of **vertices** V, and a set of **edges** E; each edge in E is a connection between a pair of vertices in V, which are called **adjacent** vartices.

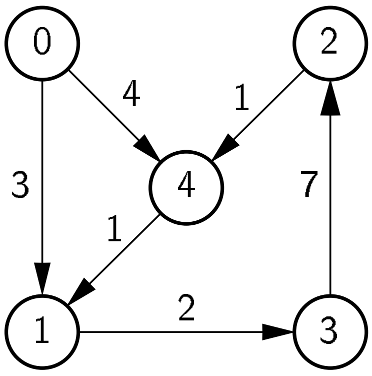
# vertices written |V|; # edges written |E|.
$0 \leq |E| \leq |V|^2$.

A graph is

- **sparse** if it has "few" edges
- **dense** if it has "many" edges
- **complete** all possible edges
- **undirected** as in figure (a)
- **directed** as in figure (b)
- **labeled** (figure (c))if it has labels on vertices
- **weighted** (figure (c))if it has (numeric) labels on edges


(a)


(b)



(c)

Graph Definitions (Cont)

A sequence of vertices $v_1, v_2, ..., v_n$ forms a **path** of length $n - 1$ ($\Rightarrow$ length = # edges) if there exist edges from v_i to v_{i+1} for $1 \leq i < n$.

A path is **simple** if all vertices on the path are distinct.

In a *directed* graph

- a path $v_1, v_2, ..., v_n$ forms a **cycle** if $n > 1$ and $v_1 = v_n$. The cycle is **simple** if, in addition, $v_2, ..., v_n$ are distinct
- a *cycle* v, v is a **self-loop**
- a directed graph with no self-loops is **simple**

In an *undirected* graph

- a path $v_1, v_2, ..., v_n$ forms a (simple) **cycle** if $n > 3$ and $v_1 = v_n$ (and, in addition, $v_2, ..., v_n$ are distinct)
  - hence the path ABA is *not* a cycle, while ABCA *is* a cycle

Graph Definitions (Cont)

Subgraph $S = (V_S, E_S)$ of a graph $G = (V, E)$: $V_S \subset V$ and $E_S \subset E$ and both vertices of any edge in E_S are in V_S

An undirected graph is **connected** if there is at least one path from any vertex to any other.

The maximal connected subgraphs of an undirected graph are called **connected components**.

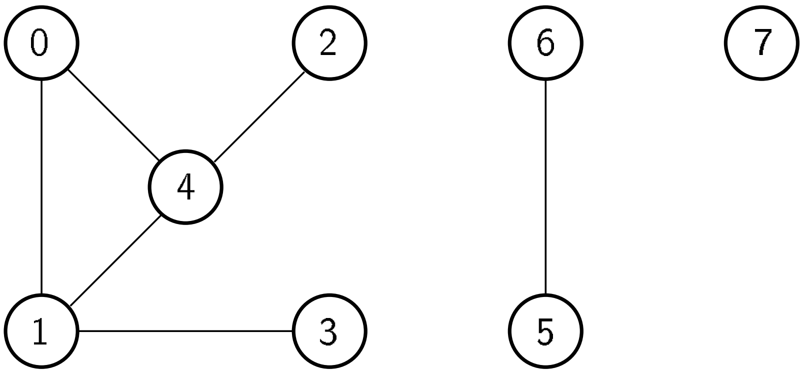
A graph without cycles is **acyclic**.

A directed graph without cycles is a **directed acyclic graph** or DAG.

A **free tree** is a connected, undirected graph with no cycles. Equivalently, a free tree is connected and has $|V - 1|$ edges.

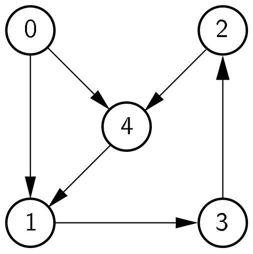
Connected Components

A graph with (composed of) 3 connected components



Graph Representations

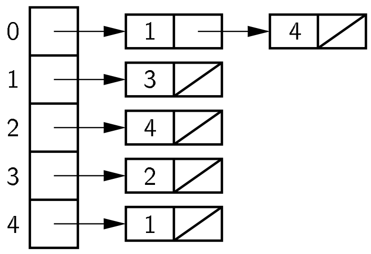
Adjacency Matrix: space required $\Theta(|V|^2)$.
Adjacency List: space required $\Theta(|V| + |E|)$.



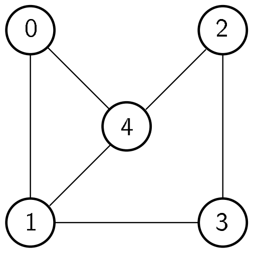
(a)

$$\begin{array}{c|ccccc|}
 & 0 & 1 & 2 & 3 & 4 \\
\hline
0 & & 1 & & & 1 \\
1 & & & & 1 & \\
2 & & & & & 1 \\
3 & & & 1 & & \\
4 & & 1 & & & \\
\hline
\end{array}$$

(b)



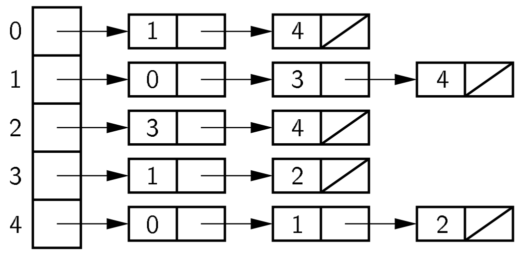
(c)



(a)

$$\begin{array}{c|ccccc|}
 & 0 & 1 & 2 & 3 & 4 \\
\hline
0 & & 1 & & & 1 \\
1 & 1 & & & 1 & 1 \\
2 & & & & 1 & 1 \\
3 & & 1 & 1 & & \\
4 & 1 & 1 & 1 & & \\
\hline
\end{array}$$

(b)



(c)

[Instead of bits, the graph could store edge, weights.]

Graph Representatiosn (cont)

Adjacency list efficient for sparse graphs (only existing edges coded)

Matrix efficient for dense graphs (no pointer overload)

Algorithms visiting each neighbor of each vertex more efficient on adjacency lists, especially for sparse graphs

Graph Interface

```
interface Graph {                     // Graph class ADT
 public int n();                     // Number of vertices
 public int e();                     // Number of edges
 // Get first edge having v as vertex v1
 public Edge first(int v);
 // Get next edge having w.v1 as the first edge
 public Edge next(Edge w);
 public boolean isEdge(Edge w);      // True if edge
 public boolean isEdge(int i, int j); // True if edge
 public int v1(Edge w);              // Where from
 public int v2(Edge w);              // Where to
 public void setEdge(int i, int j, int weight);
 public void setEdge(Edge w, int weight);
 public void delEdge(Edge w);        // Delete edge w
 public void delEdge(int i, int j); // Delete (i, j)
 public int weight(int i, int j);   // Return weight
 public int weight(Edge w);         // Return weight

 // Set Mark of vertex v
 public void setMark(int v, int val);

 // Get Mark of vertex v
 public int getMark(int v);
} // interface Graph
```

Edges have a double nature:

seen as pairs of vertices or as aggregate objects.

Vertices identified by an integer i, $0 \leq i \leq |V|$

Implementation: Edge Class

```
interface Edge {   // Interface for graph edges
  public int v1(); // Return the vertex it comes from
  public int v2(); // Return the vertex it goes to
} // interface Edge

// Edge class for Adjacency Matrix graph representation
class Edgem implements Edge {
  private int vert1, vert2; // The vertex indices

  public Edgem(int vt1, int vt2) //the constructor
    { vert1 = vt1; vert2 = vt2; }
  public int v1() { return vert1; }
  public int v2() { return vert2; }
} // class Edgem
```

Implementation: Adjacency Matrix

```
class Graphm implements Graph { // Adjacency matrix
  private int[][] matrix;       // The edge matrix
  private int numEdge;          // Number of edges
  public int[] Mark; // The mark array, initially all 0

  public Graphm(int n) {        // Constructor
    Mark = new int[n];
    matrix = new int[n][n];
    numEdge = 0;
  }

  public int n() { return Mark.length; }
  public int e() { return numEdge; }

  public Edge first(int v) { // Get first edge
    for (int i=0; i<Mark.length; i++)
      if (matrix[v][i] != 0)
        return new Edgem(v, i);
    return null;  // No edge for this vertex
  }

  public Edge next(Edge w) { // Get next edge
    if (w == null) return null;
    for (int i=w.v2()+1; i<Mark.length; i++)
      if (matrix[w.v1()][i] != 0)
        return new Edgem(w.v1(), i);
    return null;  // No next edge;
  }
```

Class Graphm implements interface Graph
Class Edgem implements interface Edge

Adjacency Matrix (cont)

```
public boolean isEdge(Edge w) { // True if an edge
  if (w == null) return false;
  else return matrix[w.v1()][w.v2()] != 0;
}

public boolean isEdge(int i, int j) // True if edge
  { return matrix[i][j] != 0; }

public int v1(Edge w) {return w.v1();} // Where from
public int v2(Edge w) {return w.v2();} // Where to

public void setEdge(int i, int j, int wt) {
  Assert.notFalse(wt!=0, "Cannot set weight to 0");
  if (matrix[i][j] == 0) numEdge++;
  matrix[i][j] = wt;
}

public void setEdge(Edge w, int weight) // Set weight
  { if (w != null) setEdge(w.v1(), w.v2(), weight); }

public void delEdge(Edge w) {       // Delete edge w
  if (w != null)
    if (matrix[w.v1()][w.v2()] != 0)
      { matrix[w.v1()][w.v2()] = 0; numEdge--; }
}

public void delEdge(int i, int j) { // Delete (i, j)
  if (matrix[i][j] != 0)
    { matrix[i][j] = 0;  numEdge--; }
}
```

NB: matrix[i][j]==0 iff there is no edge (i,j)

If there is no edge (i,j) then weight(i,j)=Integer.MAX_VALUE (INFINITY)

Adjacency Matrix (cont 2)

```
  public int weight(int i, int j) {  // Return weight
    if (matrix[i][j] == 0) return Integer.MAX_VALUE;
    else return matrix[i][j];
  }

  public int weight(Edge w) { // Return edge weight
    Assert.notNull(w,"Can’t take weight of null edge");
    if (matrix[w.v1()][w.v2()] == 0)
      return Integer.MAX_VALUE;
    else return matrix[w.v1()][w.v2()];
  }

  public void setMark(int v, int val)
    { Mark[v] = val; }

  public int getMark(int v) { return Mark[v]; }
} // class Graphm
```

Graph Traversals

Some applications require visiting every vertex in the graph exactly once.

Application may require that vertices be visited in some special order based on graph topology.

Example: Artificial Intelligence

- Problem domain consists of many “states.”
- Need to get from Start State to Goal State.
- Start and Goal are typically not directly connected.

To insure visiting all vertices:

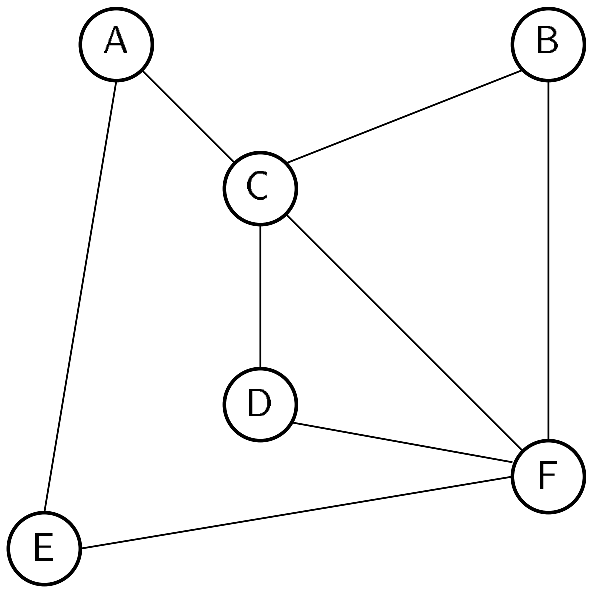
```
void graphTraverse(Graph G) {
  for (v=0; v<G.n(); v++)
    G.setMark(v, UNVISITED);  // Initialize mark bits
  //next for needed to cover all the graph in case
  //of graph composed of several connected components
  for (v=0; v<G.n(); v++)
    if (G.getMark(v) == UNVISITED)
      doTraverse(G, v);
}
```

[Two traversals we will talk about: DFS, BFS.]

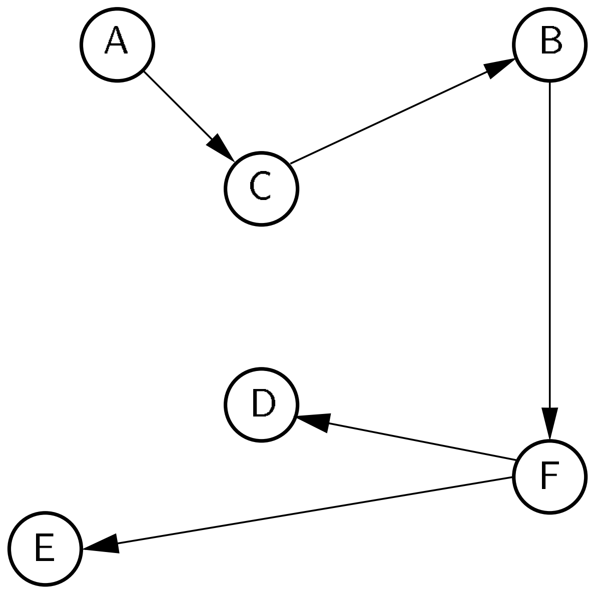
Depth First Search

```
static void DFS(Graph G, int v) { // Depth first search
  PreVisit(G, v);         // Take appropriate action
  G.setMark(v, VISITED);
  for (Edge w = G.first(v); G.isEdge(w); w = G.next(w))
    if (G.getMark(G.v2(w)) == UNVISITED)
      DFS(G, G.v2(w));
  PostVisit(G, v);        // Take appropriate action
}
```

Cost: $\Theta(|V| + |E|)$.



(a)



(b)

[The directions are imposed by the traversal. This is the Depth First Search Tree.]

If PreVisit simply prints and PostVisit does nothing then DFS prints

A C B F D E

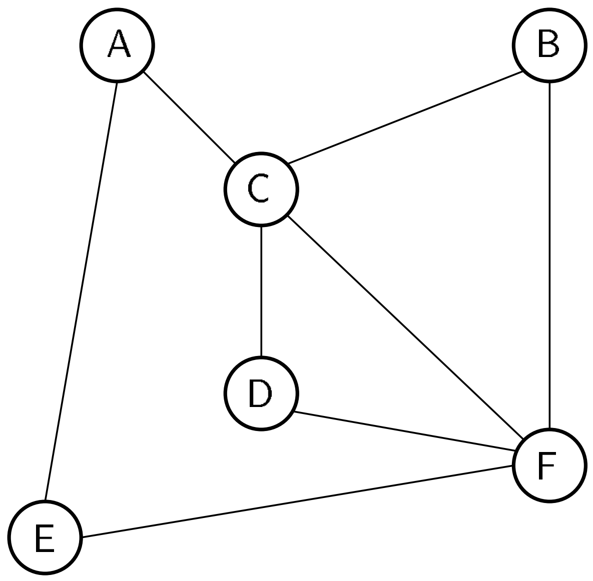
Breadth First Search

Like DFS, but replace stack with a queue.

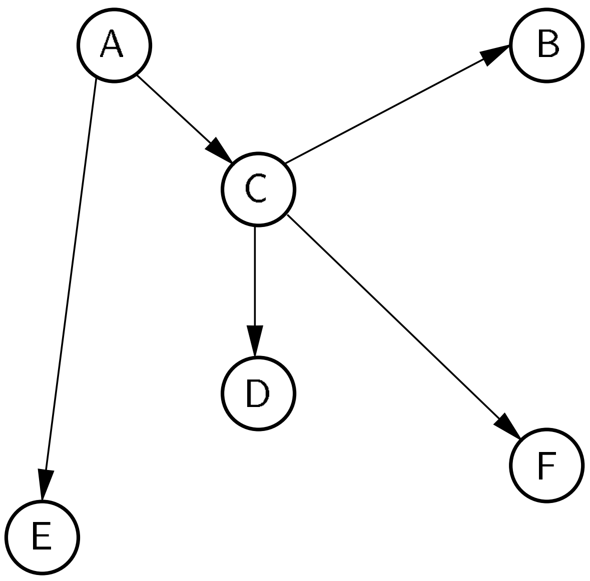
Visit the vertex's neighbors before continuing deeper in the tree.

```
static void BFS(Graph G, int start) {
  Queue Q = new AQueue(G.n());        // Use a Queue
  Q.enqueue(new Integer(start));
  G.setMark(start, VISITED);
  while (!Q.isEmpty()) { // Process each vertex on Q
    int v = ((Integer)Q.dequeue()).intValue();
    PreVisit(G, v);           // Take appropriate action
    for (Edge w=G.first(v); G.isEdge(w); w=G.next(w))
      if (G.getMark(G.v2(w)) == UNVISITED) {
        G.setMark(G.v2(w), VISITED);
        Q.enqueue(new Integer(G.v2(w)));
      }
    PostVisit(G, v);          // Take appropriate action
  }
}
```

If PreVisit simply prints and PostVisit does nothing then BFS prints A C E B D F



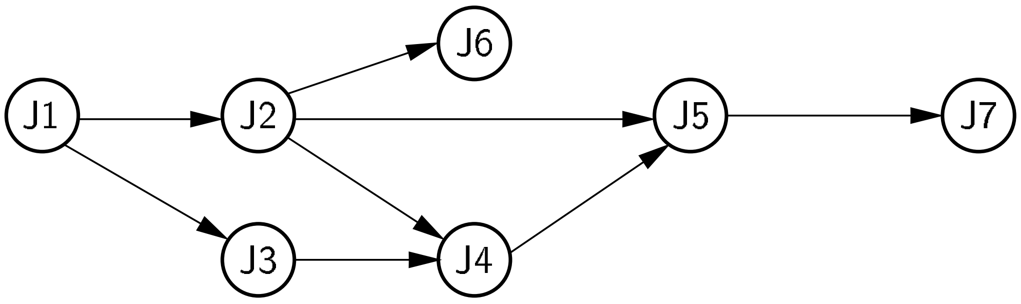
(a)



(b)

Topological Sort

Problem: Given a set of jobs, courses, etc. with prerequisite constraints, output the jobs in an order that does not violate any of the prerequisites. (NB: the graph must be a DAG)



```
static void topsort(Graph G) { // Topo sort: recursive
  for (int i=0; i<G.n(); i++)  // Initialize Mark array
    G.setMark(i, UNVISITED);
  for (int i=0; i<G.n(); i++)  // Process all vertices
    if (G.getMark(i) == UNVISITED)
      tophelp(G, i);           // Call helper function
}

static void tophelp(Graph G, int v) { // Topsort helper
  G.setMark(v, VISITED);
  for (Edge w = G.first(v); G.isEdge(w); w = G.next(w))
    if (G.getMark(G.v2(w)) == UNVISITED)
      tophelp(G, G.v2(w));
  printout(v);              // PostVisit for Vertex v
}
```

[Prints in reverse order: J7, J5, J4, J6, J2, J3, J1

It is a DFS with a PreVisit that does nothing]

Queue-based Topological Sort

```
static void topsort(Graph G) { // Topo sort: Queue
  Queue Q = new AQueue(G.n());
  int[] Count = new int[G.n()];
  int v;
  for (v=0; v<G.n(); v++) Count[v] = 0; // Initialize
  for (v=0; v<G.n(); v++)      // Process every edge
    for (Edge w=G.first(v); G.isEdge(w); w=G.next(w))
       Count[G.v2(w)]++;       // Add to v2's count
  for (v=0; v<G.n(); v++)      // Initialize Queue
    if (Count[v] == 0)         // Vertex has no prereqs
      Q.enqueue(new Integer(v));
  while (!Q.isEmpty()) {       // Process the vertices
    v = ((Integer)Q.dequeue()).intValue();
    printout(v);               // PreVisit for Vertex V
    for (Edge w=G.first(v); G.isEdge(w); w=G.next(w)) {
      Count[G.v2(w)]--;        // One less prerequisite
      if (Count[G.v2(w)] == 0) // This vertex now free
        Q.enqueue(new Integer(G.v2(w)));
    }
  }
}
```

Sorting

Each record is stored in an array and contains a field called the **key**.

Linear (i.e., total) order: comparison.

[$a < b$ **and** $b < c \Rightarrow a < c$**.]**

The Sorting Problem

Given a sequence of records $R_1, R_2, ..., R_n$ with key values $k_1, k_2, ..., k_n$, respectively, arrange the records into any order s such that records $R_{s_1}, R_{s_2}, ..., R_{s_n}$ have keys obeying the property $k_{s_1} \leq k_{s_2} \leq ... \leq k_{s_n}$.

[Put keys in ascending order.]

NB: there can be records with the same key

A sorting algorithm is **stable** if after sorting records with the same key have the same relative position as before

Measures of cost:

- Comparisons
- Swaps (when records are large)

Sorting (cont)

Assumptions: for every record type there are functions

- R.key returns the key value for record R
- DSutil.swap(array, i, j) swaps records in positions i and j of the array

Measure of the "degree of disorder" of an array in the number of **INVERSIONS**

$\forall el = a[i]$,

$\#inversions = \#elements > el$ which are in a position $j < i$

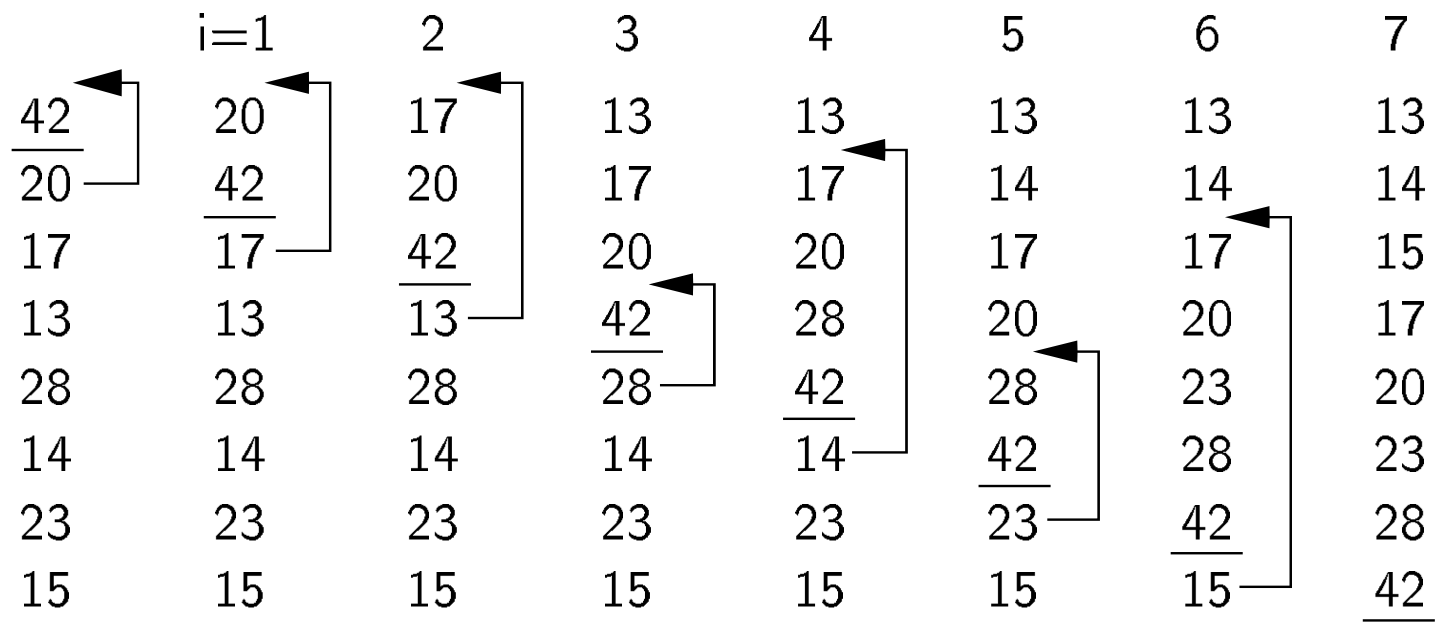
#inversions for the entire array =
$= \sum$ #inversions of each array element

For a sorted array #inversions = 0

For an array with elements in decreasing order #inversions = $\Theta(n^2)$

Insertion Sort

```
static void inssort(Elem[] array) {  // Insertion Sort
  for (int i=1; i<array.length; i++) // Insertrecord
    for (int j=i; (j>0) &&
            (array[j].key()<array[j-1].key()); j--)
      DSutil.swap(array, j, j-1);
}
```



Best Case: **[0 swaps, $n-1$ comparisons]**

Worst Case: **[$n^2/2$ swaps and compares]**

Average Case: **[$n^2/4$ swaps and compares: # inner loop iterations for an element in position n = #inversione = $n/2$ in the average]**

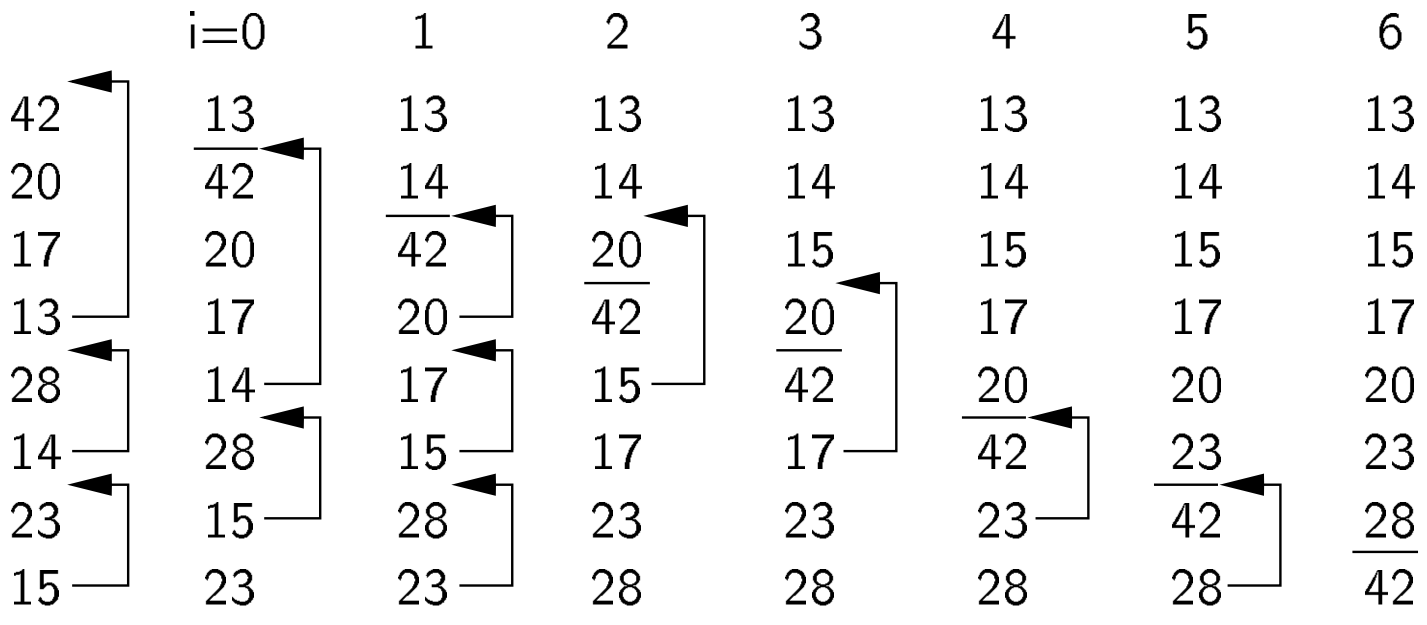
[At each iteration takes one element to its place and does only that; it works only on the sorted portion of the array]

[Nearly best performance when input "nearly sorted" $\Rightarrow$ used in conjunction with mergesort and quicksort small array segments]

Bubble Sort

```
static void bubsort(Elem[] array) {    // Bubble Sort
  for (int i=0; i<array.length-1; i++) // Bubble up
    for (int j=array.length-1; j>i; j--)
      if (array[j].key() < array[j-1].key())
        DSutil.swap(array, j, j-1);
}
```

[Using test "$j > i$" saves a factor of 2 over "$j > 0$".]



Best Case: **[$n^2/2$ compares, 0 swaps]**

Worst Case: **[$n^2/2$ compares, $n^2/2$ swaps]**

Average Case: **[$n^2/2$ compares, $n^2/4$ swaps]**

[At each iteration takes the smallest to its place, but it moves also other ones;

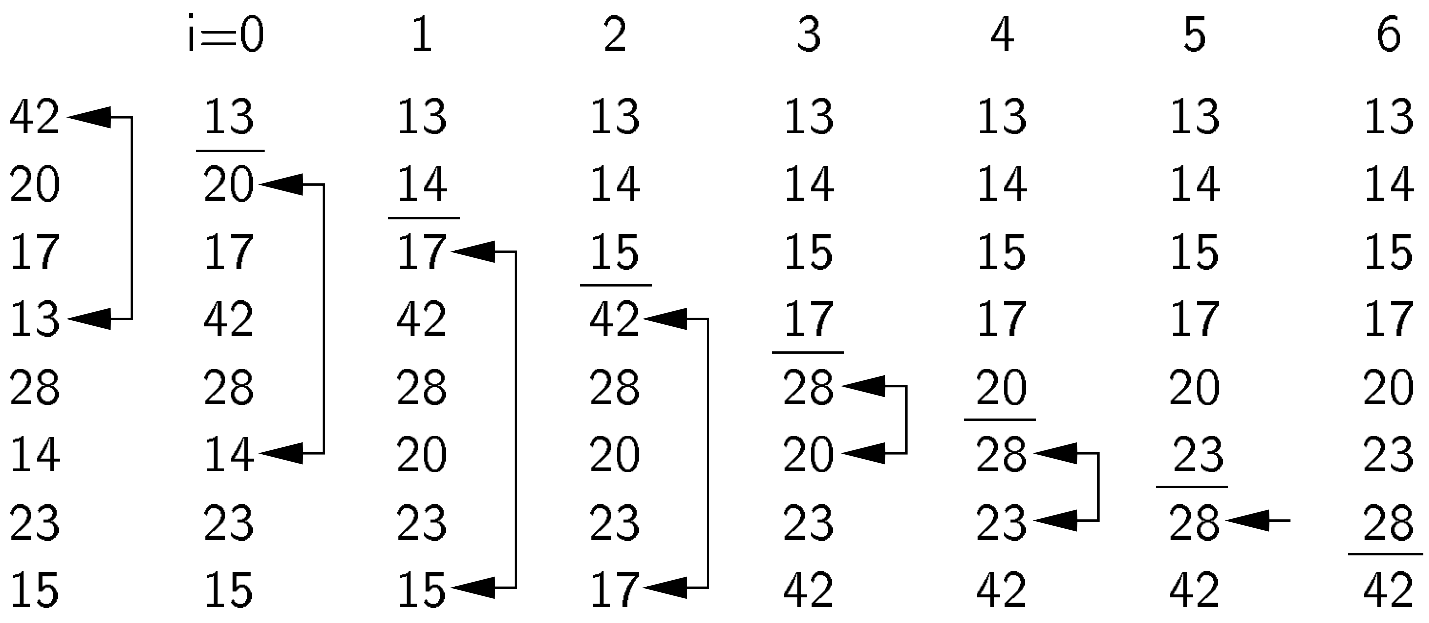
NB: it works also on the unsorted part of the array;

No redeeming features to this sort.]

Selection Sort

```
static void selsort(Elem[] array) {   // Selection Sort
  for (int i=0; i<array.length-1; i++) { // Select i'th
    int lowindex = i;              // Remember its index
    for (int j=array.length-1; j>i; j--) // Find least
      if (array[j].key() < array[lowindex].key())
        lowindex = j;              // Put it in place
    DSutil.swap(array, i, lowindex);
  }
}
```

[Select the value to go in the ith position.]



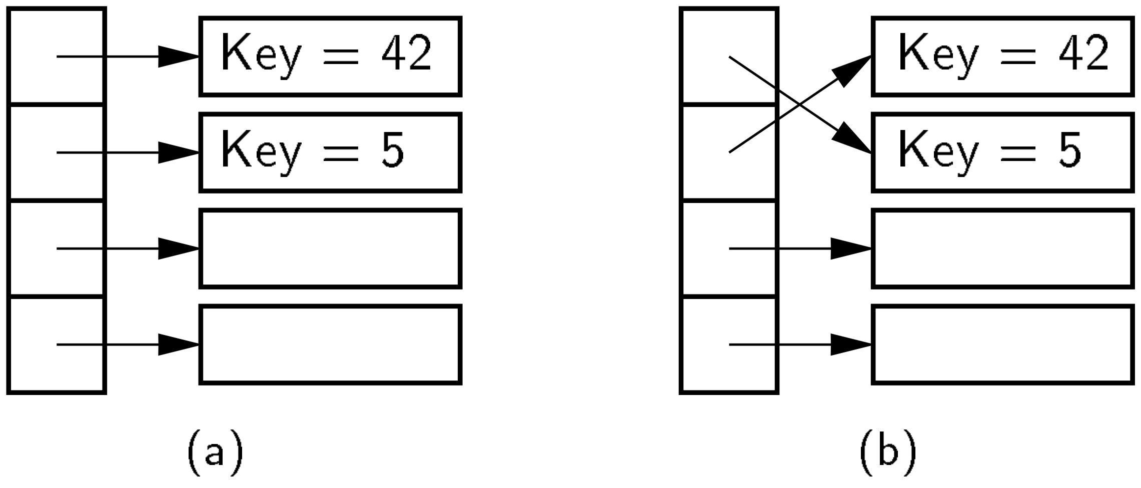
Best Case: **[0 swaps ($n-1$ as written), $n^2/2$ compares.]**

Worst Case: **[$n-1$ swaps, $n^2/2$ compares]**

Average Case: **[$O(n)$ swaps, $n^2/2$ compares]**

[It minimizes # swaps]

Pointer Swapping



[For large records.]

This is what done in Java, when records are objects

Exchange Sorting

Summary

	Insertion	Bubble	Selection
Comparisons:			
Best Case	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Swaps:			
Best Case	0	0	$\Theta(n)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$

Mergesort

```
List mergesort(List inlist) {
  if (inlist.length() <= 1) return inlist;;
  List l1 = half of the items from inlist;
  List l2 = other half of the items from inlist;
  return merge(mergesort(l1), mergesort(l2));
}
```

Analyze first the algorithm for *merging* sorted sublists

- examine first element of each sublist
- pick the smaller element (it is the smallest overall)
- remove it from its sublist and put it in the output list
- when one sublist is exhausted, pick from the other

Complexity of merging two sorted sublist: $\Theta(n)$

$$36 \quad 20 \quad 17 \quad 13 \quad 28 \quad 14 \quad 23 \quad 15$$

$$\boxed{20 \quad 36}\ \boxed{13 \quad 17}\ \boxed{14 \quad 28}\ \boxed{15 \quad 23}$$

$$\boxed{13 \quad 17 \quad 20 \quad 36}\ \boxed{14 \quad 15 \quad 23 \quad 28}$$

$$\boxed{13 \quad 14 \quad 15 \quad 17 \quad 20 \quad 23 \quad 28 \quad 36}$$

Mergesort Implementation

Mergesort is tricky to implement.
Main question: how to represent lists?

Linked lists

- merging does not require direct access, but...
- splitting requires a list traversal ($\Theta(n)$), whether list size is known (take as two sublists the first and second halves) or unknown (assign elements alternating between the two lists)

Lists represented by arrays

- splitting very easy ($\Theta(1)$) if array bounds are known
- merging easy ($\Theta(n)$) only if sub-arrays merged into a second array (hence double the space requirement!)
- avoid the need for a $distinct$ additional array for each recursive call by $first$ copying sub-arrays into auxiliary array and then merging them back to the original array (hence can use only one array for the overall process)

Mergesort Implementation (2)

```
static void mergesort(Elem[] array, Elem[] temp,
                      int l, int r) {
  if (l == r) return;              // One element list
  int mid = (l+r)/2;               // Select midpoint
  mergesort(array, temp, l, mid);  // Ssort first half
  mergesort(array, temp, mid+1, r); // Sort second half
  merge(array, temp, l, mid, mid+1, r);
}
```

```
static void merge(Elem[] array, Elem[] temp,
                  int l1, int r1, int l2, int r2) {
  for (int i=l1; i<=r2; i++)       // Copy subarrays
    temp[i] = array[i];
  // Do the merge operation back to array
  int i1 = l1; int i2 = l2;
  for (int curr=l1; curr<=r2; curr++) {
    if (i1 > r1)  // Left sublist exhausted
      array[curr] = temp[i2++];
    else if (i2 > r2)  // Right sublist exhausted
      array[curr] = temp[i1++];
    // else choose least of the two front elements
    else if (temp[i1].key() < temp[i2].key())
      array[curr] = temp[i1++]; // Get smaller val
    else array[curr] = temp[i2++];
  }
}
```

Complexity of Mergesort

Ad hoc analysis

- depth of recursion is $\log n$
- at each recursion depth i
  - 2^i recursive calls
  - each recursive call has array length $n/2^i$, hence...
  - total length of merged arrays is n at every depth
- therefore total cost is $\mathsf{T}(n) = \Theta(n \log n)$

Alternative analysis: use recurrence equation

$\mathsf{T}(n) = a\mathsf{T}(n/b)+cn^k = 2\mathsf{T}(n/2)+cn$, $\mathsf{T}(1)=d$

We have $a = b = 2$, $k = 1$ and therefore $a = b^k$, hence

$\mathsf{T}(n) = \Theta(n \log n)$

Heapsort

Heapsort uses a max-heap.

```
static void heapsort(Elem[] array) {  // Heapsort
  MaxHeap H = new MaxHeap(array, array.length,
                          array.length);
  for (int i=0; i<array.length; i++)  // Now sort
    H.removemax();   // Put max value at end of heap
}
```

Cost of Heapsort: **[$\Theta(n \log n)$]**

Cost of finding k largest elements: **[$\Theta(k \log n + n)$.**

Time to build heap: $\Theta(n)$.

Time to remove least element: $\Theta(\log n)$.]

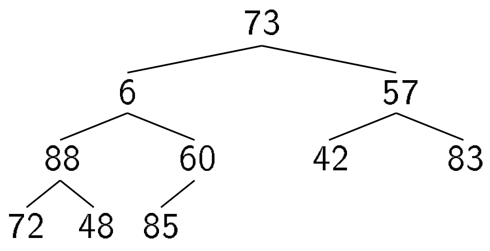
[Compare to sorting with BST: this is expensive in space (overhead), potential bad balance, BST does not take advantage of having all records available in advance.]

[Heap is space efficient, balanced, and building initial heap is efficient.]

Heapsort Example

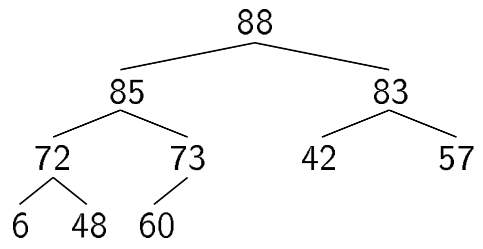
Original Numbers

73	6	57	88	60	42	83	72	48	85



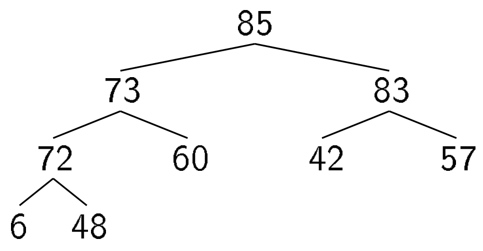
Build Heap

88	85	83	72	73	42	57	6	48	60



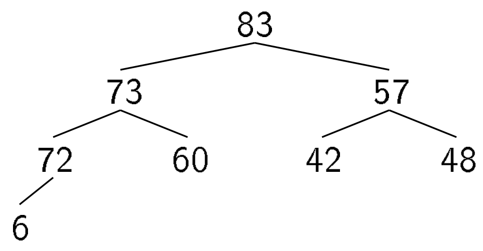
Remove 88

85	73	83	72	60	42	57	6	48	88



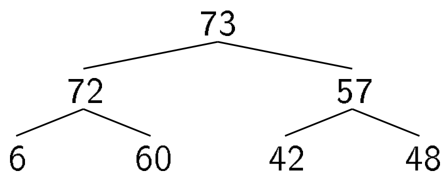
Remove 85

83	73	57	72	60	42	48	6	85	88



Remove 83

73	72	57	6	60	42	48	83	85	88



Empirical Comparison

[MS Windows – CISC]

Algorithm	**10**	**100**	**1000**	**10,000**
Insert. Sort	.10	9.5	957.9	98,086
Bubble Sort	.13	14.3	1470.3	157,230
Select. Sort	.11	9.9	1018.9	104,897
Shellsort	.09	2.5	45.6	829
Quicksort	.15	1.8	23.6	291
Quicksort/O	.10	1.6	20.9	274
Mergesort	.12	2.4	36.8	505
Mergesort/O	.08	1.8	28.0	390
Heapsort	–	50.0	60.0	880
Radix Sort/1	.87	8.6	89.5	939
Radix Sort/4	.23	2.3	22.5	236
Radix Sort/8	.19	1.2	11.5	115

[UNIX – RISC]

Algorithm	**10**	**100**	**1000**	**10,000**
Insert. Sort	.66	65.9	6423	661,711
Bubble Sort	.90	85.5	8447	1,068,268
Select. Sort	.73	67.4	6678	668,056
Shellsort	.62	18.5	321	5,593
Quicksort	.92	12.7	169	1,836
Quicksort/O	.65	10.7	141	1,781
Mergesort	.76	16.8	234	3,231
Mergesort/O	.53	11.8	189	2,649
Heapsort	–	41.0	565	7,973
Radix Sort/1	7.40	67.4	679	6,895
Radix Sort/4	2.10	18.7	160	1,678
Radix Sort/8	4.10	11.5	97	808

[Clearly, $n \log n$ superior to n^2. Note relative differences on different machines.]

Upperbound and Lowerbound for a Problem

Upperbound: asymptotic cost of the fastest $known$ algorithm

lowerbound best possible efficiency of any possible (known or unknown) algorithm

open problem: upperbound different from (greater than) lowerbound

closed problem: upperbound equal to lowerbound

Sorting Lower Bound

Want to prove a lower bound sorting problem based on key comparison.

Sorting I/O takes $\Omega(n)$ time. (no algorithm can take less than I/O time)

Sorting is $O(n \log n)$.

Will now prove $\Omega(n \log n)$ lower bound.

Form of proof:

- Comparison based sorting can be modeled by a binary tree.
- The tree must have $\Omega(n!)$ leaves (because there are $n!$ permutations of n elements).
- The tree cannot be less than $\Omega(n \log n)$ levels deep (a tree with k nodes has at least $\log k$ levels).

this comes from the fact that

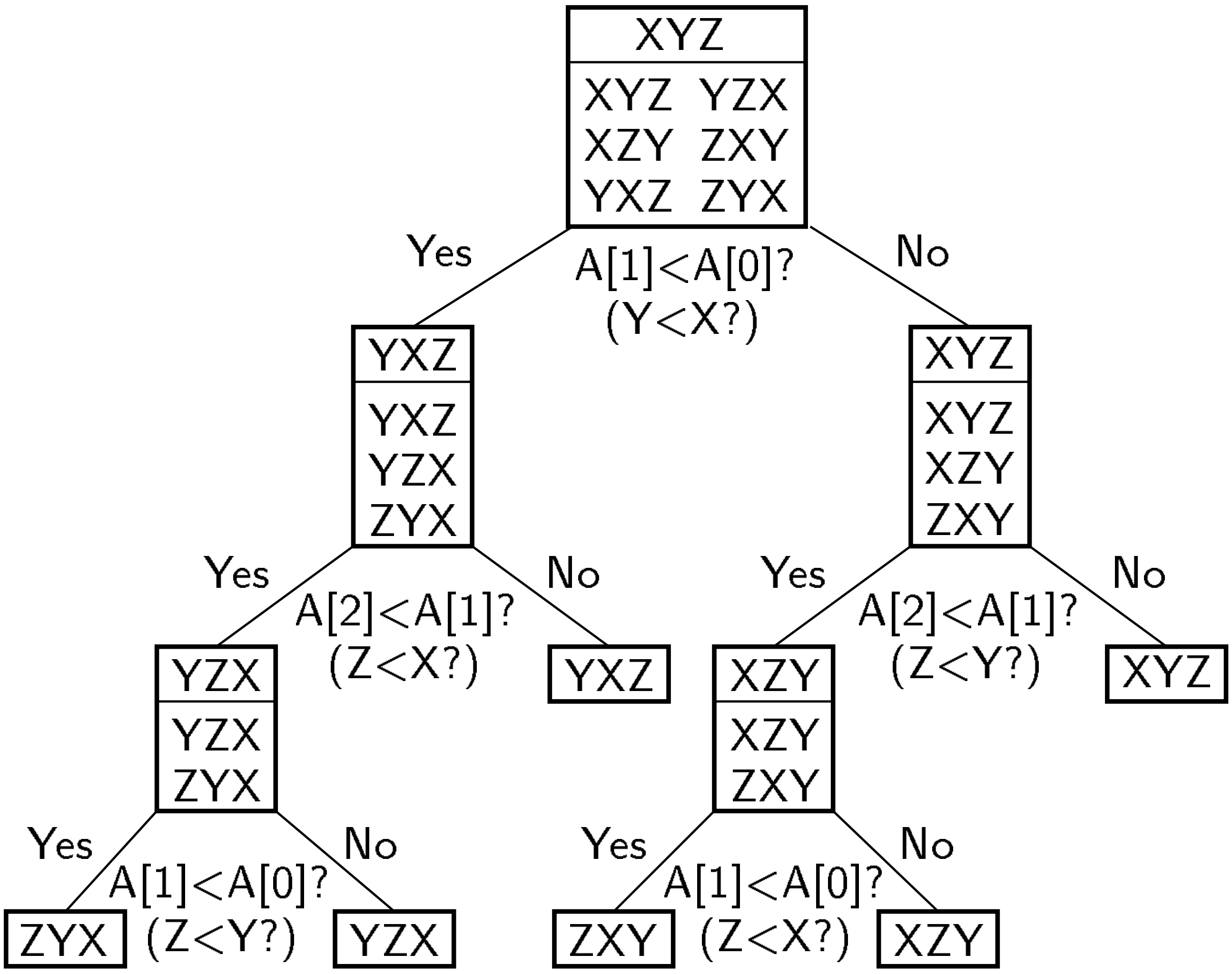
$$\log n! = \Theta(n \log n)$$

which is due to Stirling's approximation of $n!$:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

from which $\log n! \approx n \log n$

Decision Trees



[Illustration of Insertion sort. Lower part of table shows possible output (sorted version of input array) after each check]

There are $n!$ permutations, and at least 1 node for each permutation.

Where is the worst case in the decision tree?

Primary vs. Secondary Storage

review following sections of textbook

9.1 on Primary vs secondary storage

9.2 on Disk & Tape drives

9.3 on Buffers and Buffer Pools

Buffer Pools

A series of buffers used by an application to cache disk data is called a **buffer pool**.

Virtual memory uses a buffer pool to imitate greater RAM memory by actually storing information on disk and "swapping" between disk and RAM.

Caching Same technique to imitate greater CACHE memory by storing info on RAM and swapping between RAM and CACHE.

Organization for buffer pools: which one to use next?

- First-in, First-out: Use the first one on the queue.
- Least Frequently Used (LFU): Count buffer accesses, pick the least used.
- Least Recently Used (LRU):
  Keep buffers on linked list.
  When a buffer is accessed, bring to front.
  Reuse the one at the end.

Programmer's View of Files

Logical view of files:

- An array of bytes.
- A **file pointer** marks the current position.

Three fundamental operations:

- Read bytes from current position (move file pointer).
- Write bytes to current position (move file pointer).
- Set file pointer to specified byte position.

Java File Functions

```
RandomAccessFile(String name, String mode)

close()

read(byte[] b)

write(byte[] b)

seek(long pos)
```

External Sorting

Problem: Sorting data sets too large to fit in main memory.

- Assume data stored on disk drive.

To sort, portions of the data must be brought into main memory, processed, and returned to disk.

An external sort should minimize disk accesses.

Model of External Computation

Secondary memory is divided into equal-sized **blocks** (512, 2048, 4096 or 8192 bytes are typical sizes).

The basic I/O operation transfers the contents of one disk block to/from main memory.

Under certain circumstances, reading blocks of a file in sequential order is more efficient. (When?) **[1) Adjacent logical blocks of file are physically adjacent on disk. 2) No competition for I/O head.]**

Typically, the time to perform a single block I/O operation is sufficient to Quicksort the contents of the block.

Thus, our primary goal is to minimize the number of block I/O operations.

Most workstations today must do all sorting on a single disk drive.

Key Sorting

Often records are large while keys are small.

- Ex: Payroll entries keyed on ID number.

Approach 1: Read in entire records, sort them, then write them out again.

Approach 2: Read only the key values, store with each key the location on disk of its associated record.

If necessary, after the keys are sorted the records can be read and re-written in sorted order.

[But, this is not usually done. (1) It is expensive (random access to all records). (2) If there are multiple keys, there is no "correct" order.]

External Sort: Simple Mergesort

Quicksort requires random access to the entire set of records.

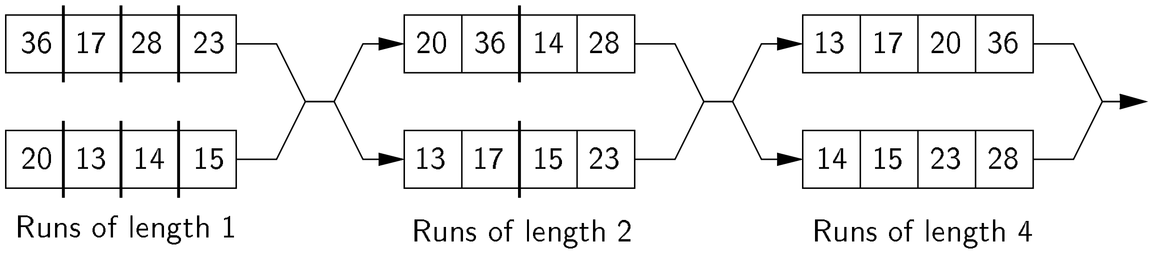
Better: Modified Mergesort algorithm

- Process n elements in $\Theta(\log n)$ passes.

1. Split the file into two files.
2. Read in a block from each file.
3. Take first record from each block, output them in sorted order.
4. Take next record from each block, output them to a *second* file in sorted order.
5. Repeat until finished, alternating between output files. Read new input blocks as needed.
6. Repeat steps 2-5, except this time the input files have groups of two sorted records that are merged together.
7. Each pass through the files provides larger and larger groups of sorted records.

A group of sorted records is called a **run**.

Problems with Simple Mergesort



Is each pass through input and output files sequential? **[yes]**

What happens if all work is done on a single disk drive? **[Competition for I/O head eliminates advantage of sequential processing.]**

How can we reduce the number of Mergesort passes? **[Read in a block (or several blocks) and do an in-memory sort to generate large initial runs.]**

In general, external sorting consists of two phases:

1. Break the file into initial runs.
2. Merge the runs together into a single sorted run.

Breaking a file into runs

General approach:

- Read as much of the file into memory as possible.
- Perform and in-memory sort.
- Output this group of records as a single run.

General Principals of External Sorting

In summary, a good external sorting algorithm will seek to do the following:

- Make the initial runs as long as possible.
- At all stages, overlap input, processing and output as much as possible.
- Use as much working memory as possible. Applying more memory usually speeds processing.
- If possible, use additional disk drives for more overlapping of processing with I/O, and allow for more sequential file processing.

Search

Given: Distinct keys k_1, k_2, ... k_n and collection T of n records of the form

$$(k_1, I_1), (k_2, I_2), ..., (k_n, I_n)$$

where I_j is information associated with key k_j for $1 \leq j \leq n$.

Search Problem: For key value K, locate the record (k_j, I_j) in T such that $k_j = K$.

Exact match query: search records with a specified key value.

Range query: search records with key in a specified range.

Searching is a systematic method for locating the record (or records) with key value $k_j = K$.

A **successful** search is one in which a record with key $k_j = K$ is found.

An **unsuccessful** search is one in which no record with $k_j = K$ is found (and presumably no such record exists).

Approaches to Search

1. Sequential and list methods (lists, tables, arrays).
2. Direct access by key value (hashing).
3. Tree indexing methods.

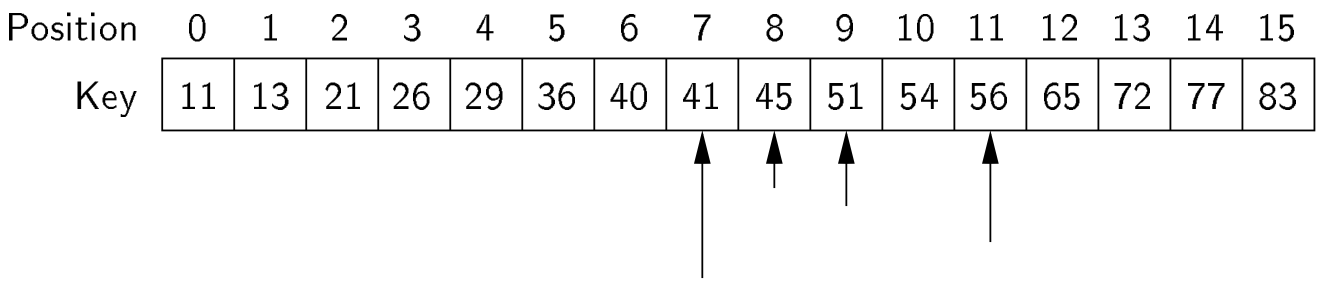
[recall: sequences: duplicate key values allowed; sets no key duplication]

Searching Ordered Arrays

Sequential Search

Binary Search

```
static int binary(int K, int[] array,
                  int left, int right) {
  // Return position of element (if any) with value K
  int l = left-1;
  int r = right+1;   // l and r are beyond array bounds
  while (l+1 != r) { // Stop when l and r meet
    int i = (l+r)/2; // Look at middle of subarray
    if (K < array[i]) r = i;     // In left half
    if (K == array[i]) return i; // Found it
    if (K > array[i]) l = i;     // In right half
  }
  return UNSUCCESSFUL; // Search value not in array
}
```


Position 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Key 11 13 21 26 29 36 40 41 45 51 54 56 65 72 77 83

Improvement: **Dictionary Search**, expected record position computed from key value; value of key found there used as in binary search

Lists Ordered by Frequency

Order lists by (expected) frequency of occurrence $\Rightarrow$ Perform sequential search.

Cost to access first record: 1; second record: 2

Expected (i.e., average) search cost:

$$\overline{C}_n = 1p_1 + 2p_2 + ... + np_n$$

[p_i is probability of ith record being accessed.]

Example: all records have equal frequency

$$\overline{C}_n = \sum_{i=1}^{n} i/n = (n+1)/2.$$

Example: Exponential frequency

$$p_i = \begin{cases} 1/2^i & \text{if } 1 \leq i \leq n-1 \\ 1/2^{n-1} & \text{if } i = n \end{cases}$$

[Second line is to make proabilities sum to 1.]

$$\overline{C}_n \approx \sum_{i=1}^{n} (i/2^i) \approx 2.$$

[very good performance, because assumption (exp. freq.) is strong]

Zipf Distributions

Applications:

- Distribution for frequency of word usage in natural languages.
- Distribution for populations of cities, etc.

Definition: Zipf frequency for item i in the distribution for n records as $1/i\mathcal{H}_n$.

[$\mathcal{H}_n = \sum_{i=1}^{n} \frac{1}{i} \approx \log_e n$**.]**

$$\overline{C}_n = \sum_{i=1}^{n} i/i\mathcal{H}_n = n/\mathcal{H}_n \approx n/\log_e n$$

80/20 rule: 80% of the accesses are to 20% of the records.

For distributions following the 80/20 rule,

$$\overline{C}_n \approx 0.122n.$$

Self-Organizing Lists

Self-organizing lists modify the order of records within the list based on the actual pattern of record access.

Based on assumption that past searches provide good indication of future ones

This is a **heuristic** similar to those for managing buffer pools.

- Order by actual historical frequency of access. (Similar to LFU buffer pool replacement strategy.) **[COUNT method: slow reaction to change]**
- **Move-to-Front**: When a record is found, move it to the front of the list. **[Not worse than twice "best arrangement"; easy to implement with linked lists, not arrays]**
- **Transpose**: When a record is found, swap it with the record ahead of it. **[A pathological, though unusual case: keep swapping last two elements.]**

Advantages of self-organizing lists

- do not require sorting
- cost of insertion and deletion low
- no additional space
- simple (hence easy to implement)

Example of Self-Organizing Tables

Application: Text compression.

Keep a table of words already seen, organized via Move-to-Front Heuristic.

If a word not yet seen, send the word.
Otherwise, send the (current) index in the table.

[NB: sender and receiver maintain identical lists, so they agree on indices]

The car on the left hit the car I left.

The car on 3 left hit 3 5 I 5.

Hashing

Hashing: The process of mapping a key value to a position in a table.

A **hash function** maps key values to positions It is denoted by **h**.

A **hash table** is an array that holds the records. It is denoted by T.

[NB: records not necessarily ordered by key value or frequency]

The hash table has M slots, indexed from 0 to $M - 1$.

For any value K in the key range and some hash function **h**,

$\mathbf{h}(K) = i, 0 \leq i < M$, such that $T[i]$.key() $= K$.

Hashing (continued)

Hashing is appropriate only for sets (no duplicates).

Good for both in-memory and disk based applications.

[Very good for organizing large databases on disk]

Answers the question "What record, if any, has key value K?"

[Not good for range queries.]

Trivial Example: Store the n records with keys in range 0 to $n-1$.

- Store the record with key i in slot i.
- Use hash function $\mathbf{h}(K) = K$.

Typically, there are however many more values in the key range than slots in the hash table

Collisions

More reasonable example:

- Store about 1000 records with keys in range 0 to 16,383.
- Impractical to keep a hash table with 16,384 slots.
- We must devise a hash function to map the key range to a smaller table.

Given: hash function **h** and keys k_1 and k_2.
β is a slot in the hash table.
If $\mathbf{h}(k_1) = \beta = \mathbf{h}(k_2)$, then k_1 and k_2 have a **collision** at β under **h**.

Perfect Hashing: hash function devised so that there are no collisions

Often impractical, sometimes expensive but worthwhile

It works when the set is **very stable** (e.g., a database on a CD)

Collisions (cont)

Search for the record with key K:

1. Compute the table location $\mathbf{h}(K)$.
2. Starting with slot $\mathbf{h}(K)$, locate the record containing key K using (if necessary) a **collision resolution policy**.

Collisions are inevitable in most applications.

- Example: 23 people are likely to share a birthday ($p = \frac{1}{2}$).

Example: store 200 students, in a table T with 365 records, using hash function h: birthday

Hash Functions

A hash function MUST return a value within the hash table range.

To be practical, a hash function SHOULD evenly distribute the records stored among the hash table slots.

Ideally, the hash function should distribute records with equal probability to all hash table slots. In practice, success depends on the distribution of the actual records stored.

If we know nothing about the incoming key distribution, evenly distribute the key range over the hash table slots.

If we have knowlege of the incoming distribution, use a distribution-dependant hash function.

Hash Functions (cont.)

Reasons why data values are poorly distributed

- Natural distributions are exponential (e.g., populations of cities)
- collected (e.g., measured) values are often somehows skewed (e.g., rounding when measuring)
- coding and alphabets introduce uneven distributions (e.g., words in natural language have first letter poorly distributed)

Example Hash Functions

```
static int h(int x) {
  return(x % 16);
}
```

This function is entirely dependant on the lower 4 bits of the key, likely to be poorly distributed.

Mid-square method: square the key value, take the middle r bits from the result for a hash table of 2^r slots.

[Works well because all bits contribute to the result.]

Sum the ASCII values of the letters and take results modulo M.

```
static int h(String x, int M) {
  int i, sum;
  for (sum=0, i=0; i<x.length(); i++)
    sum += (int)x.charAt(i);
  return(sum % M);
}
```

[Only good if the sum is large compared to the size of the table M.]

[This is an example of a folding method]

[NB: order of characters in the string is immaterial]

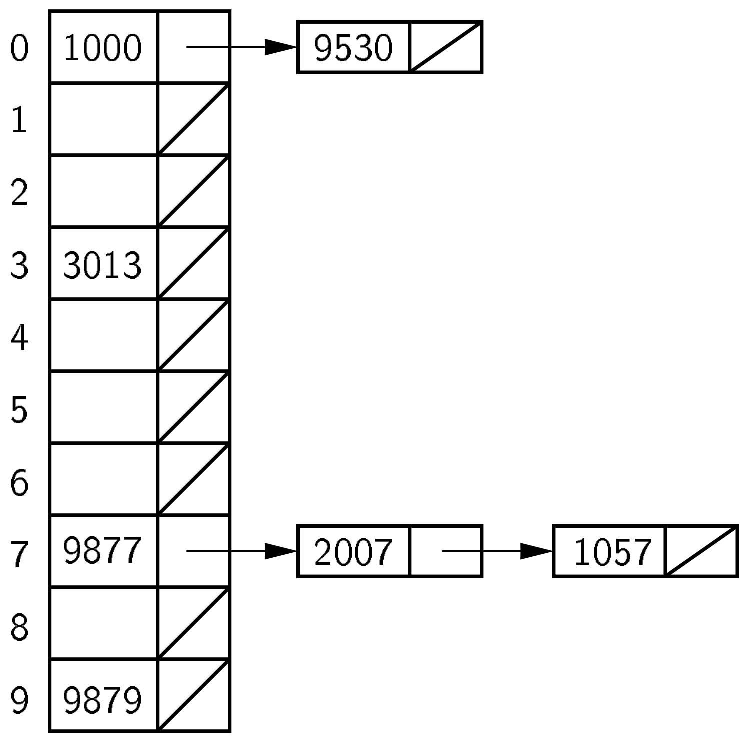
Open Hashing

What to do when collisions occur?

Open hashing treats each hash table slot as a bin.

Open: collisions result in storing values outside the table

Each slot is the head of a linked list



Open Hashig Performance

Factors influencing performance

- how records are ordered in a slot's list (e.g., by key value or frequency of access)
- ration N/M (records/slots)
- distribution of record key values

NB: Open Hash table must be kept in main memory (storing on disk would defeat the purpose of hashing)

Bucket Hashing

Divide the hash table slots into buckets.

- Example: 8 slots/bucket.

Include an overflow bucket.

Records hash to the first slot of the bucket, and fill bucket. Go to overflow if necessary.

When searching, first check the proper bucket. Then check the overflow.

Closed Hashing

Closed hashing stores all records directly in the hash table.

Each record i has a **home position** $\mathbf{h}(k_i)$.

If i is to be inserted and another record already occupies i's home position, then another slot must be found to store i.

The new slot is found by a **collision resolution policy**.

Search must follow the same policy to find records not in their home slots.

Collision Resolution

During insertion, the goal of collision resolution is to find a free slot in the table.

Probe Sequence: the series of slots visited during insert/search by following a collision resolution policy.

Let $\beta_0 = \mathbf{h}(K)$. Let $(\beta_0, \beta_1, ...)$ be the series of slots making up the probe sequence.

```
void hashInsert(Elem R) { // Insert R into hash table T
  int home;               // Home position for R
  int pos = home = h(R.key());// Initial pos on sequence
  for (int i=1; T[pos] != null; i++) {
    // Find next slot: p() is the probe function
    pos = (home + p(R.key()), i)) % M;
    Assert.notFalse(T[pos].key() != R.key(),
               "Duplicates not allowed");
  }
  T[pos] = R;                 // Insert R
}
```

Collision Resolution (cont.)

```
// p(K, i) probe function returns offset from home position
// for ith slot of probe sequence of K
ELEM hashSearch(int K) { // Search for record w/ key K
  int home;              // Home position for K
  int pos = home = h(K); // Initial pos on sequence
  for (int i = 1; (T[pos] != null) &&
                  T[pos].key() != K); i++)
    pos = (home + p(K, i)) % M; // Next pos on sequence
  if (T[pos] == null) return null;  // K not in hash table
  else return T[pos];  // Found it
}
```

Linear Probing

Use the probe function

```
int p(int K, int i) { return i; }
```

This is called **linear probing**.

Linear probing simply goes to the next slot in the table.

If the bottom is reached, wrap around to the top.

To avoid an infinite loop, one slot in the table must always be empty.

Linear Probing Example

Assuming hash function $h(x) = x \bmod 11$

0	1001
1	9537
2	3016
3	
4	
5	
6	
7	9874
8	2009
9	9875
10	

(a)

0	1001
1	9537
2	3016
3	
4	
5	
6	
7	9874
8	2009
9	9875
10	1052

(b)

Primary Clustering: Records tend to cluster in the table under linear probing since the probabilities for which slot to use next are not the same.

[notation: prob(x) is the probability that next element goes to position x]

[For (a): prob(3) = 4/11, prob(4) = 1/11, prob(5) = 1/11, prob(6) = 1/11, prob(10) = 4/11.]

[For (b): prob(3) = 8/11, prob(4,5,6) = 1/11 each.]

[small clusters tend to merge $\Rightarrow$ long probe sequences]

Improved Linear Probing

Instead of going to the next slot, skip by some constant c.

Warning: Pick M and c carefully.

The probe sequence SHOULD cycle through all slots of the table.

[If $M = 10$ with $c = 2$, then we effectively have created 2 hash tables (evens vs. odds).]

Pick c to be relatively prime to M.

There is still some clustering.

- Example: $c = 2$. $\mathbf{h}(k_1) = 3$. $\mathbf{h}(k_2) = 5$.
- The probe sequences for k_1 and k_2 are linked together.

Pseudo Random Probing

The ideal probe function would select the next slot on the probe sequence at random.

An actual probe function cannot operate randomly. (Why?)

[Execution of random procedure cannot be duplicated when searching]

Pseudo random probing:

- Select a (random) permutation of the numbers from 1 to $M - 1$:

$$r_1, r_2, \ldots, r_{M-1}$$

- All insertions and searches use the same permutation.

Example: Hash table of size $M = 101$

- $r_1 = 2, r_2 = 5, r_3 = 32.$
- $\mathbf{h}(k_1) = 30, \mathbf{h}(k_2) = 28.$
- Probe sequence for k_1 is: **[30, 32, 35, 62]**
- Probe sequence for k_2 is: **[28, 30, 33, 60]**

[The two probe sequences diverge immediately]

Quadratic Probing

Set the i'th value in the probe sequence as

$$(\mathbf{h}(K) + i^2) \bmod M.$$

Example: M = 101.

- $\mathbf{h}(k_1) = 30$, $\mathbf{h}(k_2) = 29$.
- Probe sequence for k_1 is:

  [30, 31, 34, 39] $=$

  [30, 30+1$^2, 30 + 2^2, 30 + 3^2]$
- Probe sequence for k_2 is:

  [29, 30, 33, 38] $=$

  [29, 29+1$^2, 29 + 2^2, 29 + 3^2] =$

Problem: not all slots in the hash table are necessarily in the probe serquence

Double Hashing

Pseudo random probing eliminates primary clustering.

If two keys hash to same slot, they follow the same probe sequence. This is called **secondary clustering**.

To avoid secondary clustering, need a probe sequence to be a function of the original key value, not just the home position.

Double hashing:

$$\mathbf{p}(K, i) = i * \mathbf{h}_2(K) \text{ for } 0 \leq i \leq M - 1. \,]$$

Be sure that all probe sequence constants are relatively prime to M **[just like in improved linear probing]** .

Example: Hash table of size $M = 101$

- $\mathbf{h}(k_1) = 30, \mathbf{h}(k_2) = 28, \mathbf{h}(k_3) = 30.$
- $\mathbf{h}_2(k_1) = 2, \mathbf{h}_2(k_2) = 5, \mathbf{h}_2(k_3) = 5.$
- Probe sequence for k_1 is: **[30, 32, 34, 36]**
- Probe sequence for k_2 is: **[28, 33, 38, 43]**
- Probe sequence for k_3 is: **[30, 35, 40, 45]**

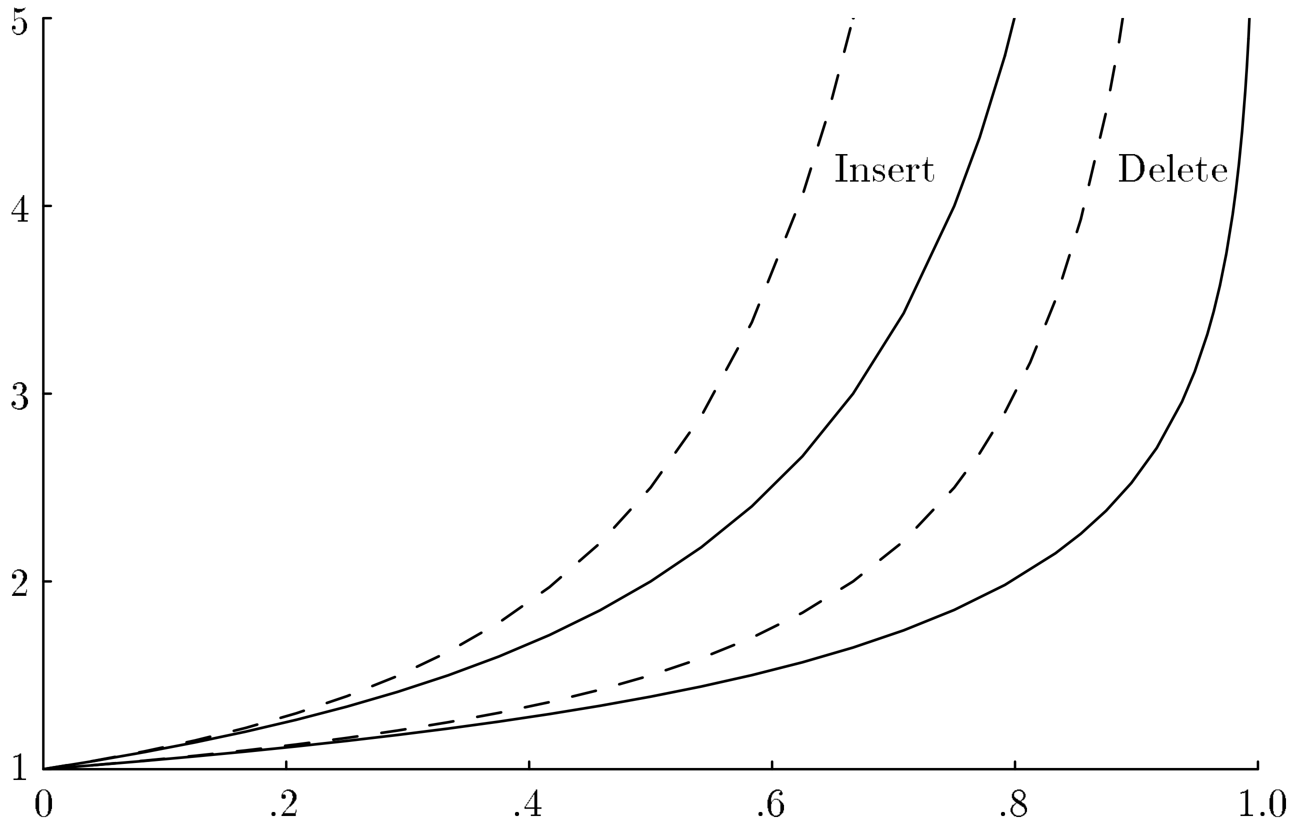
Analysis of Closed Hashing

The expected cost of hashing is a function of how full the table is

The <u>**load factor**</u> is $\alpha = N/M$ where N is the number of records currently in the table.

Expected # accesses (NB: accesses are due to collisions) vs α

- solid lines: random probing
- dashed lines: linear probing



Deletion

1. Deleting a record must not hinder later searches.
2. We do not want to make positions in the hash table unusable because of deletion.

Both of these problems can be resolved by placing a special mark in place of the deleted record, called a **tombstone**.

A tombstone will not stop a search, but that slot can be used for future insertions.

Unfortunately, tombstones do add to the average path length.

Solutions:

1. Local reorganizations to try to shorten the average path length.
2. Periodically rehash the table (by order of most frequently accessed record).

Indexing

Goals:

- Store large files.
- Support multiple search keys.
- Support efficient insert, delete and range queries.

Entry sequenced file: Order records by time of insertion. **[Not practical as a database organization.]**

Use sequential search.

Index file: Organized, stores pointers to actual records. **[Could be a tree or other data structure.]**

Primary key: A unique identifier for records. May be inconvenient for search.

Secondary key: an alternate search key, often not unique for each record. Often used for search key.

Linear Indexing

Linear Index: an index file organized as a simple sequence of key/record pointer pairs where the key values are in sorted order.

Features:

- If the index is too large to fit in main memory, a second level index may be used.
- Linear indexing is good for searching variable length records.
- Linear indexing is poor for insert/delete.

Tree Indexing

Linear index is poor for insertion/deletion.

Tree index can efficiently support all desired operations (typical of a database):

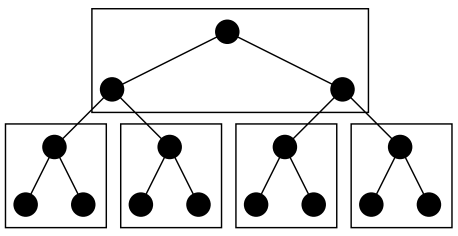
- Insert/delete
- Multiple search keys **[Multiple tree indices.]**
- Key range search

Storing a (BST) tree index on disk causes additional problems:

1. Tree *must* be balanced. **[Minimize disk accesses.]**
2. Each path from root to a leaf should cover few disk pages.

Use buffer pool to store recently accessed pages; exploit locality of reference

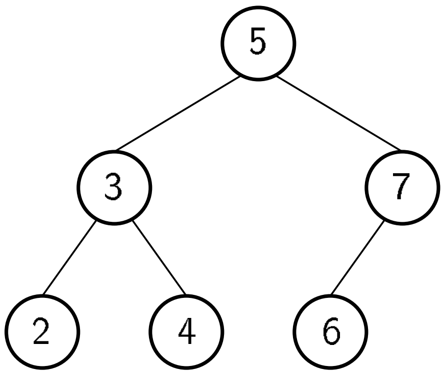
But only mitigates the problem



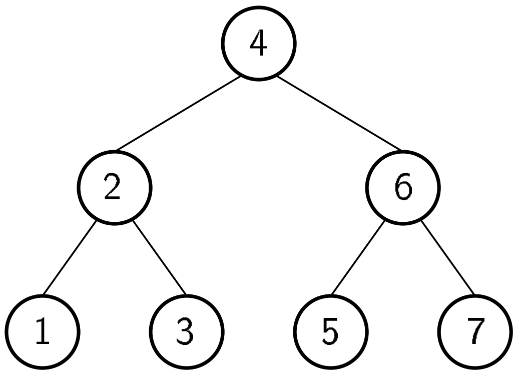
Tree indexing (cont.)

Rebalance a BST after insertion/deletion can require much rearranging

Example of insert(1)



(a)



(b)

2-3 Tree

A 2-3 Tree has the following shape properties:

1. A node contains one or two keys.
2. Every internal node has either two children (if it contains one key) or three children (if it contains two keys).
3. All leaves are at the same level in the tree, so the tree is always height balanced.

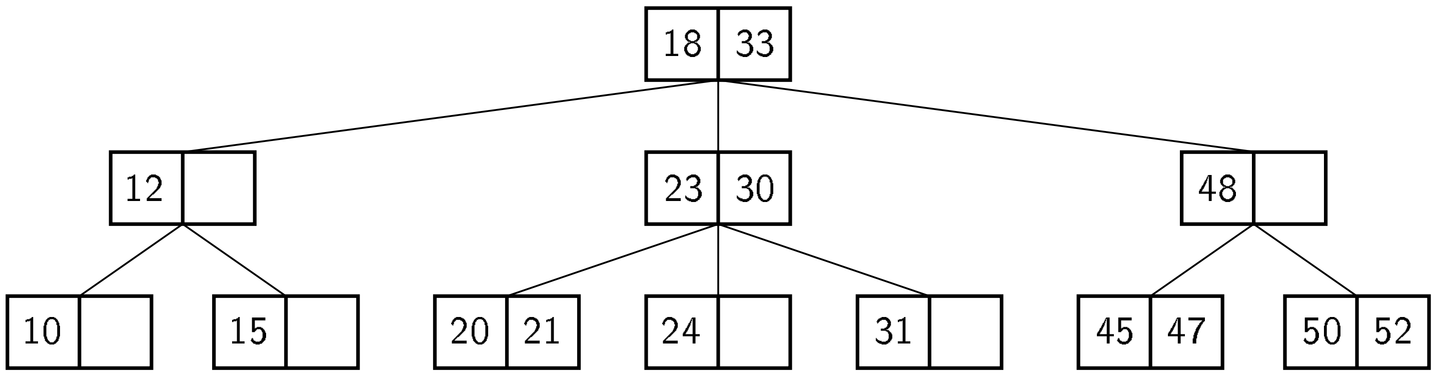
The 2-3 Tree also has search tree properties analogous to BST

1. values in left subtree $<$ first node value
2. values in center subtree $\geq$ first node value
3. values in center subtree $<$ second node value (if existing)
4. (if both existing) values in right subtree $\geq$ first node value

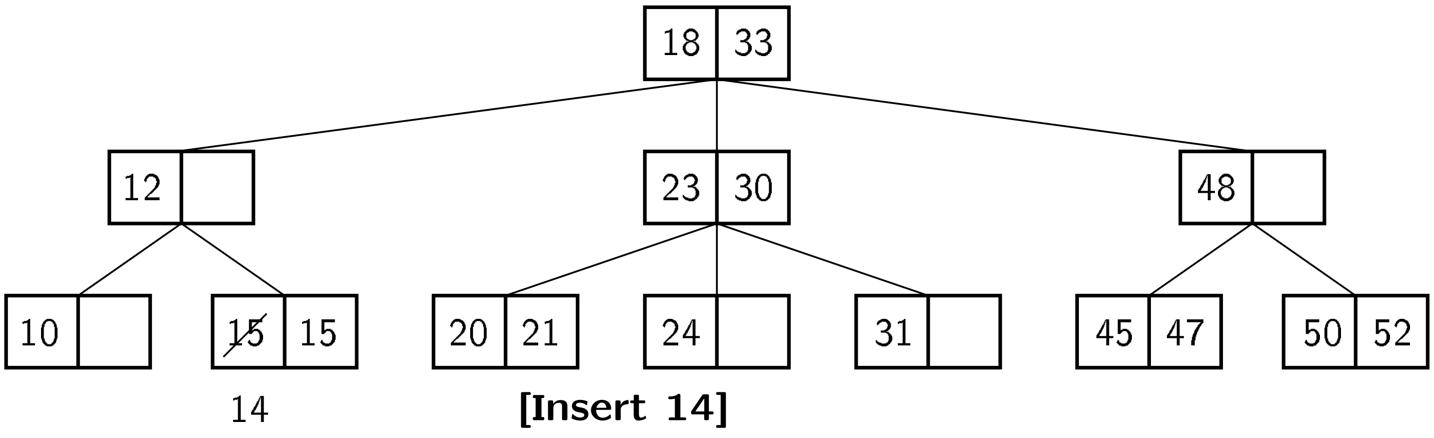
2-3 Tree(cont.)

The advantage of the 2-3 Treeover the BST is that it can be updated at low cost.

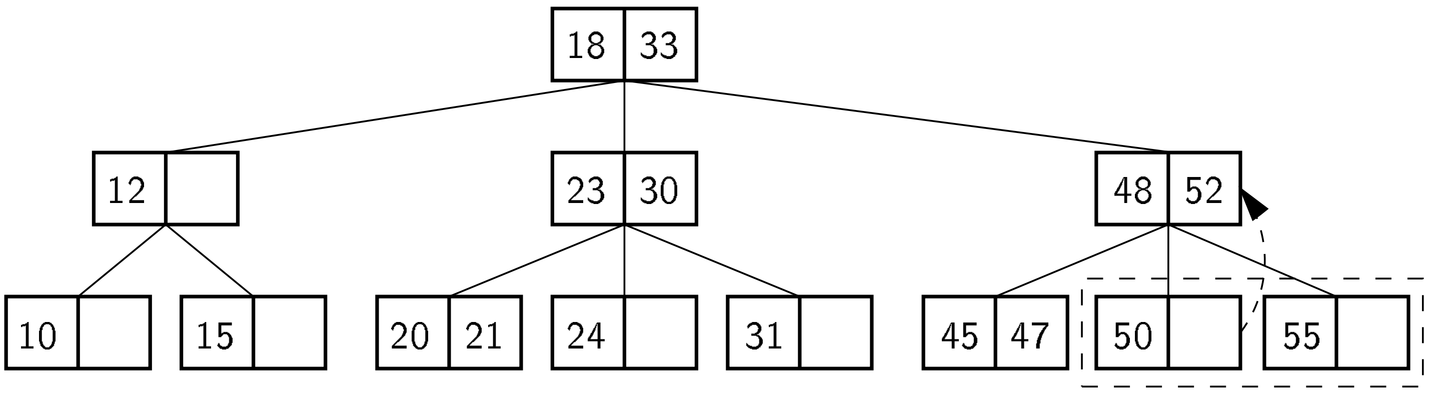
- always insert at leaf node
- search position for key to be inserted
- if there is room (1 free slot) then finished
- otherwise must add a node (split operation)
- from 1 node with 2 keys get 2 nodes with 1 key and **promote** middle valued key
- recursively, insert promoted key into parent node
- if splitting repeated until root of the tree then its depth increases (but tree remains balanced)



2-3 Tree Insertion



[Insert 14]

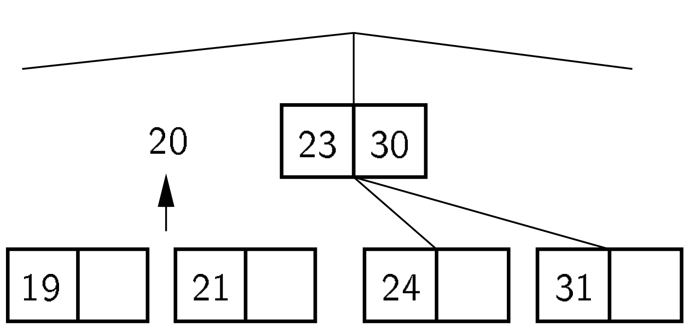


[Insert 55. Always insert at leaf node.]

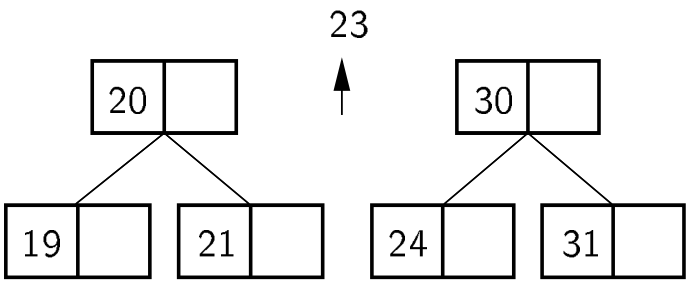
2-3 Tree Splitting

[Insert 19 into node 20-21 ⇒ split and promote 20 into node 23-30 ⇒ split and promote 23, this becomes new root, tree is 1 level deeper]

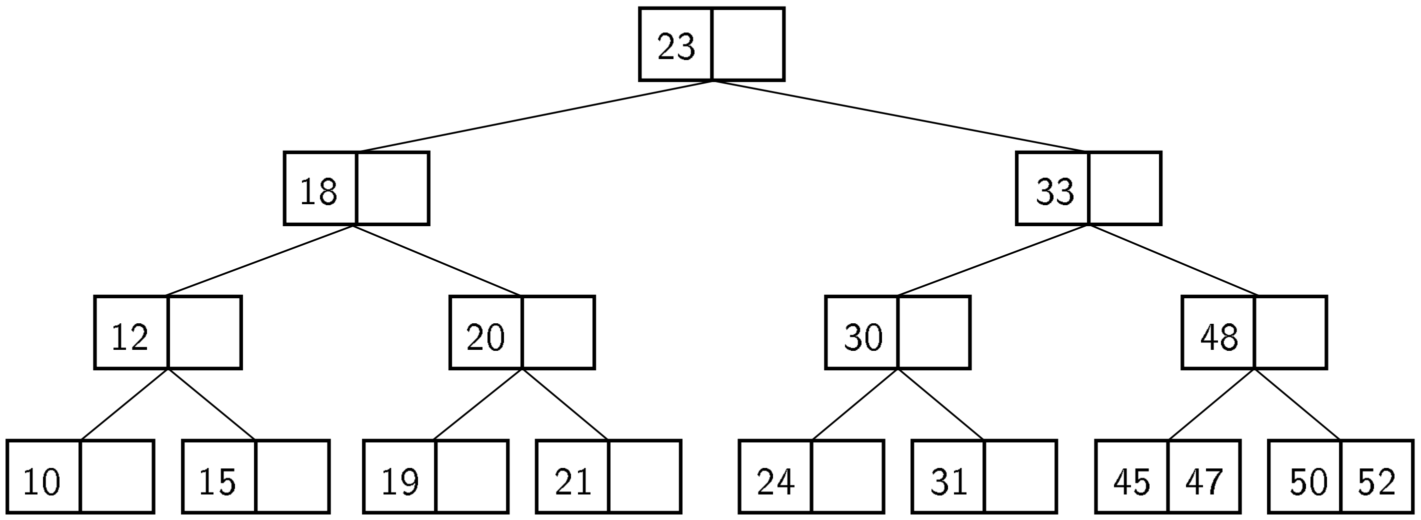
[NB: All operations are local to original search path.]



(a)



(b)



(c)

B-Trees

The B-Tree is a generalization of the 2-3 Tree.

The B-Tree is now **the** standard file organization for applications requiring insertion, deletion and key range searches.

1. B-Trees are always balanced.
2. B-Trees keep related records on a disk page, which takes advantage of locality of reference.
3. B-Trees guarantee that every node in the tree will be full at least to a certain minimum percentage. This improves space efficiency while reducing the typical number of disk fetches necessary during a search or update operation.

B-Trees (Continued)

A B-Tree of order m has the following properties.

- The root is either a leaf or has at least two children.
- Each node, except for the root and the leaves, has between $\lceil m/2 \rceil$ and m children.
- All leaves are at the same level in the tree, so the tree is always height balanced.

NB: A 2-3 Tree is a B-Tree of order 3

A B-Tree node is usually selected to match the size of a disk block.

A B-Tree node could have hundreds of children $\Rightarrow$ depth is $\approx \log_{100} n$.

A block implemented in a disk block

A pointer implemented by a disk block reference

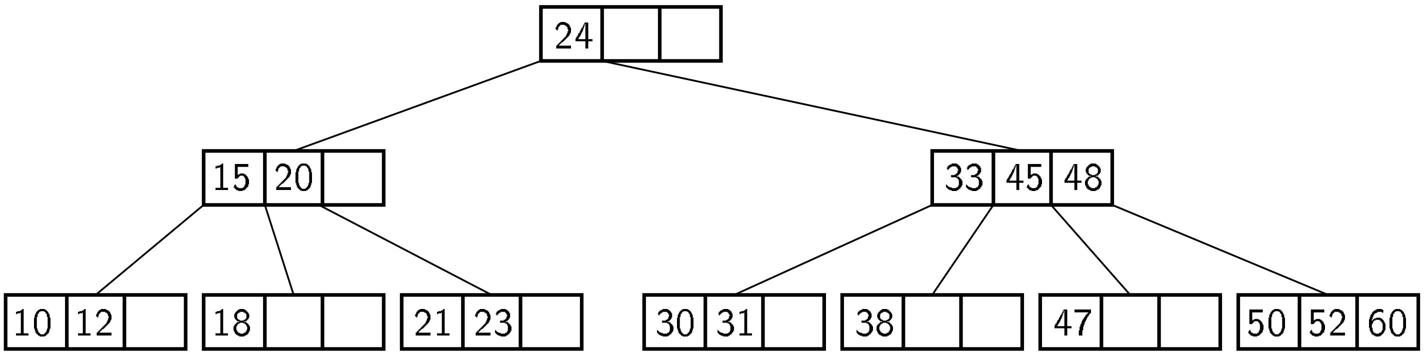
B-Tree Example

Search in a B-Tree is a generalization of search in a 2-3 Tree.

1. Perform a binary search on the keys in the current node. If the search key is found, then return the record. If the current node is a leaf node and the key is not found, then report an unsuccessful search.
2. Otherwise, follow the proper branch and repeat the process.

A B-Tree of order 4

Example: search for record with key 47



B-Tree Insertion

Obvious extension of 2-3 Tree insertion

NB: split and promote process ensures all nodes are half full

Example: node with 4 keys + add one key

SPLIT $\Rightarrow$ promote middle key + 2 nodes with 2 keys each

B^+-Trees

The most commonly implemented form of the B-Tree is the B^+-Tree.

Internal nodes of the B^+-Tree do not store records – only key values to guide the search.

Leaf nodes store records or pointers to records.

A leaf node may store more or less records than an internal node stores keys.

Requirement: leaf nodes always half full.

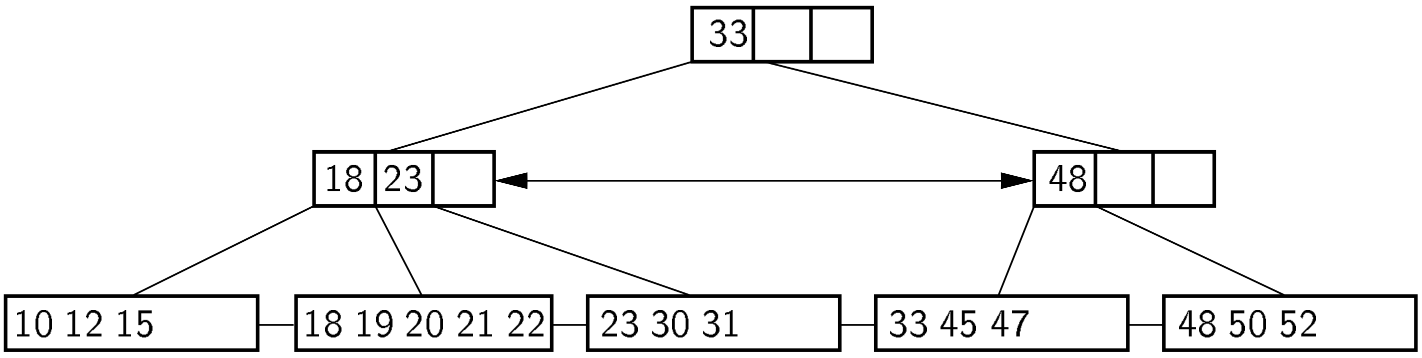
Leaf nodes doubly linked in a list $\Rightarrow$ can traverse it in any order $\Rightarrow$ very good for range queries.

Search: similar to B-Tree search: must always go to the leaf (internal nodes do not store records)

B^+-Tree Example: search

[Assume leaves can store 5 values, internal notes 3 (4 children).]

[Example: search key 33]



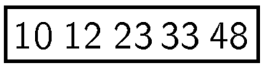
B$^+$-Tree Insertion

Insertion similar to B-Tree insertion:

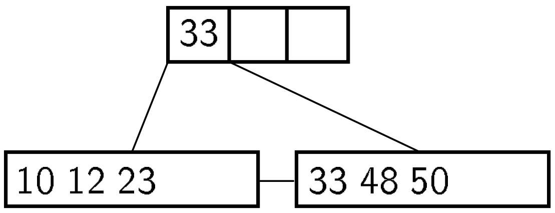
- find leaf that should contain inserted key
- if not full, insert and finish
- else split and promote *a copy* of least valued key of the newly formed right node

B^+-Tree Example: Insertion

[Note special rule for root: May have only two children.]

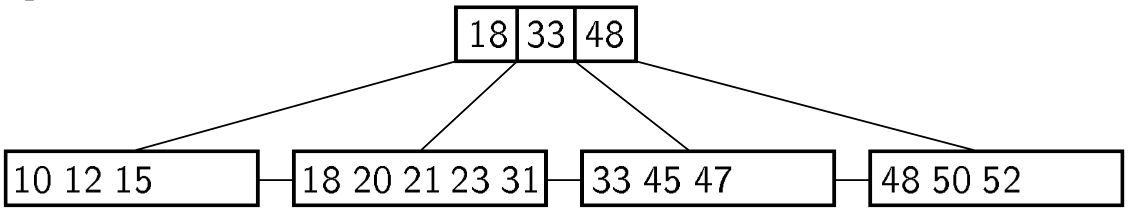


(a)



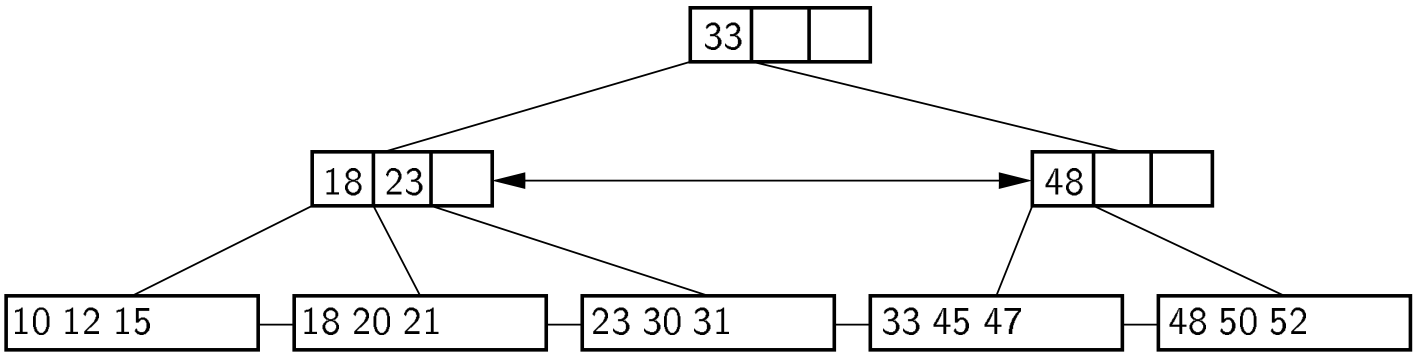
(b)

[(b) Add 50.] **[Add 45, 52, 47 (split), 18, 15, 31 (split), 21, 20.]**



(c)

[Add 30 (split).]



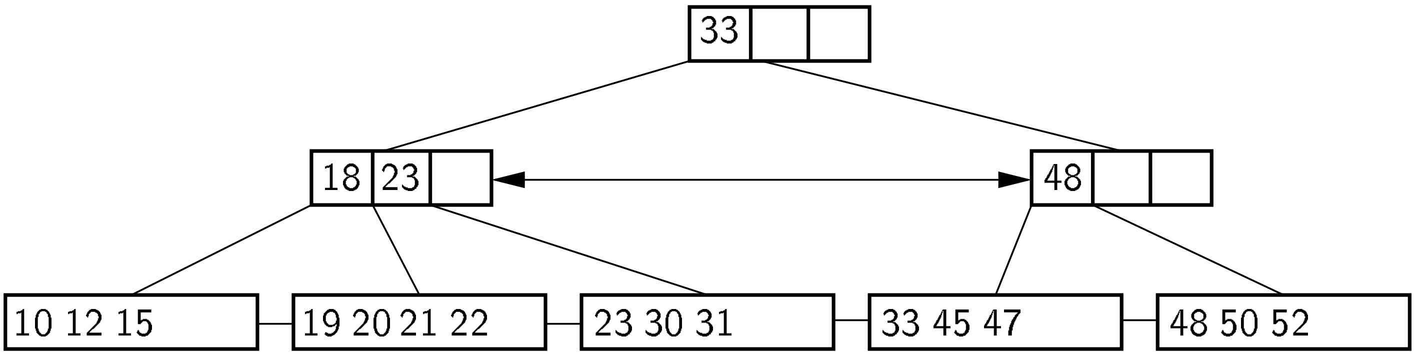
(d)

B^+-Tree Deletion

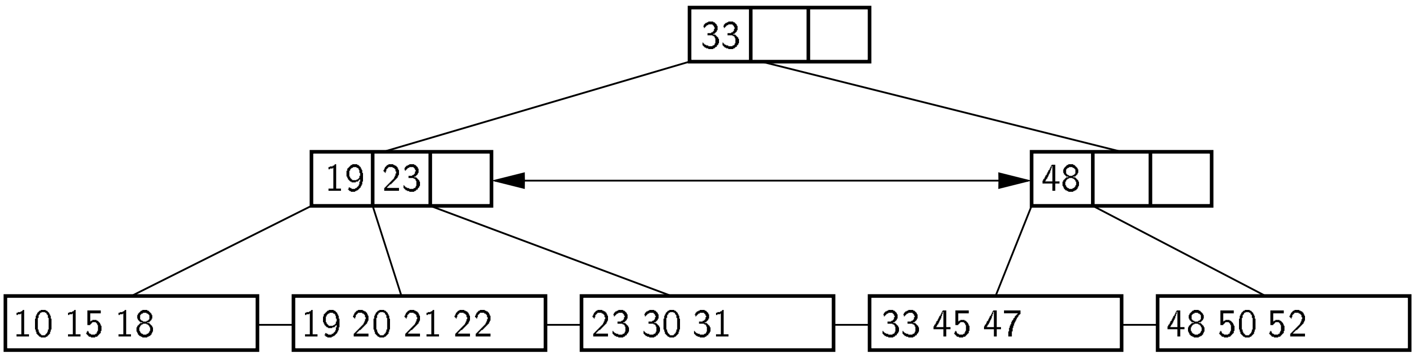
- locate level N containing key to be deleted
- if more than half full, remove and finish
- else (underflow) must restructure the tree
  - if possible get spare values from adjacent siblings ($\Rightarrow$ possibly keys in parent node must be updated)
  - if siblings cannot give values (they are only half full)
    - N goves its values to them and is removed (possible because its siblings are only half full and N is underflowing)
    - this can cause underflow in the parent node ($\Rightarrow$ propagate upwards, possibly eventually causing two chindren or root to merge and tree to lose one level)

B^+-Tree Example: Deletion

[Simple delete – delete 18 from original example.] [NB: do not need to delete 18 from internal node: it is a placeholder, can still be used to guide the search]



[Delete of 12 form original example: Borrow from sibling.]



B-Tree Space Analysis

B$^+$-Tree nodes are always at least half full.

The B*-Tree splits two pages for three, and combines three pages into two. In this way, nodes are always 2/3 full.

Improves performance, makes implementation *very complex*

Tradeoff between space utilization and efficiency and complexity of impolementation

Asymptotic cost of search, insertion and deletion of records from B-Trees, B$^+$-Trees and B*-Trees is $\Theta(\log n)$. (The base of the log is the (average) branching factor of the tree.)

Ways to reduce the number of disk fetches:

- Keep the upper levels in main memory.
- Manage B$^+$-Tree pages with a buffer pool.

B-Tree Space Analysis: Examples

Example: Consider a B$^+$-Tree of order 100 with leaf nodes containing 100 records.

1 level B$^+$-Tree: **[Max: 100]**

2 level B$^+$-Tree: **[Min: 2 leaves of 50 for 100 records. Max: 100 leaves with 100 for 10,000 records.]**

3 level B$^+$-Tree: **[Min:** 2×50 **nodes of leaves for 5000 records. Max:** $100^3 = 1,000,000$ **records.]**

4 level B$^+$-Tree: **[Min: 250,000 records (2 * 50 * 50 * 50). Max: 100 million records (100 * 100 * 100 * 100).]**