Advanced Programming Laboratory

Hema A Murthy

Indian Institute of Technology Madras, India.

September 6, 2021

Lists

The ADT List:
A sequence of zero or more elements of a given *ElementType*:

$$a_1, a_2, a_3, ..., a_n$$
$$n \geq 0 \text{ and } a_i \text{ is of } ElementType$$
$$n - \text{is the length of the List}$$
$$n \geq 1, \text{ first } element\ is\ a_1\ and \text{ last } is\ a_n$$

Examples for Lists:

- A list of jobs
- A list grocery items
- Representation of sets
- Lists can be used to perform infinite precision arithmetic.

The ADT List I

ADT List { private Data:
a - list of ElementType

- insert (x, p) - insert element x at position p
- end() - got to the end of the list
- locate(x) - find the position of element x in List
- retrieve(p) - retrieve the element at position p
- delete(p) - delete the element at position p
- next(p) - get the position of element at the position next to p
- prev(p) - get the position of element at the position before p
- makeNull() - create an empty list
- first() - get the position of the first element
- print() - print all the elements in the list
- print(p) - print the element at position p

}

The ADT List II

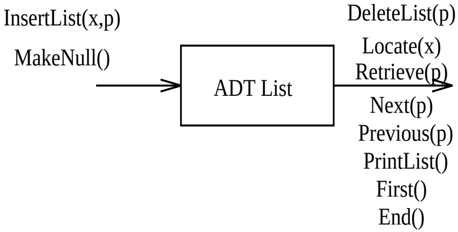


Figure: The List ADT

Lists I

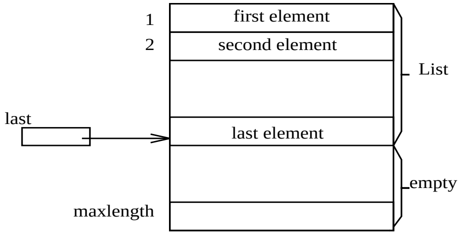


Figure: The List ADT - array implementation

Lists II

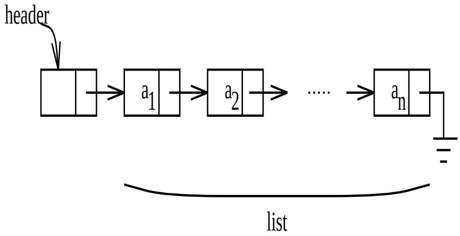


Figure: The List ADT - linked list implementation

Implementation of Lists I

```
/***************************************************
 * Program              : LinkList.cxx
 * Function             : Defines a class List
 ***************************************************/

#include "iostream.h"
/* Definition of element of List */

typedef struct CellType* Position;
typedef int ElemType;
struct CellType {
  ElemType value;
  Position next;
};
```

Implementation of Lists II

```
/*Definiton of class List */
class List { /* begin {Definition of class List} */
private:
  Position listHead;                         //pointer to hea
public:
  void makeNull();                           // create a new
  void insertList(ElemType x, Position p); // insert x at P
  void deleteList(Position p);               // delete elemen
  Position first();                          // get Position
  Position end();                            // get Position
  Position next(Position p);                 // get Position
  void printList();                          // print List
}; /* end {Definition of class List} */
```

Implementation of Lists III

```
/* begin {Implementation of class List} */

void List::makeNull() {
  listHead = new CellType;
  listHead->next = NULL;
}


void List::insertList(ElemType x, Position p) {
  Position          temp;
  temp = p->next;
  p->next = new CellType;
  p->next->next  = temp;
  p->next->value = x;
}
```

Implementation of Lists IV

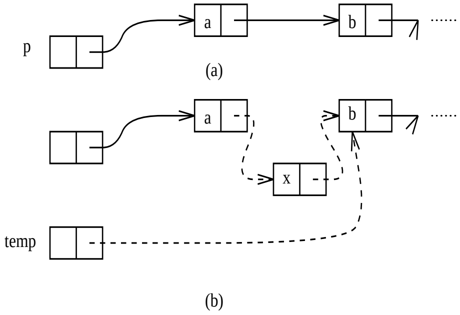


Figure: The List ADT - Insertion Operation

Implementation of Lists V

```
void List::DeleteList(Position p) {
    p->next = p->next->next;
}
```

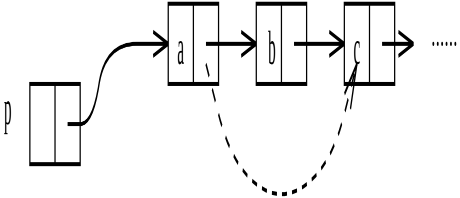


Figure: The List ADT - Deletion Operation

Implementation of Lists VI

```
Position List::First() {
  return(listHead);
}

Position List::end() {
  Position                p;
  p = listHead;
  while (p->next != NULL)
    p = p->next;
  return(p);
}

Position List::next(Position p) {
  return (p->next);
}
```

Implementation of Lists VII

```
void List::printList() {
  Position            p;
  p = listHead->next;
  while (p != NULL) {
    cout << p->value << " ";
    p = p->next;
  }
  cout << endl;
}
```

Implementation of Lists VIII

List Problems

- Storing Sets, performing operations on Sets (union, intersection, set A - set B)
- Infinite precision arithmetic – the entire number is represented in a list, perform operations of +,-,*,/.

Assignment:

1. Write a function to remove duplicates in a give List using only List operations. The function takes argument a list and returns the purged list.
2. Write a function convert a given unordered list into an ordered list.

Stacks I

Definition: A special ADT to which addition and deletions are made only at one end, LIFO (last-in-first-out).

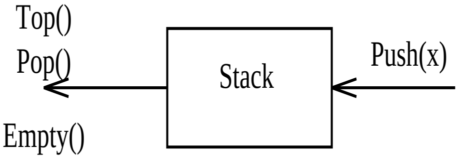


Figure: The Stack ADT

Stacks II

Application of Stacks:Conversion of infix expressions to postfix.

- While (not end of input) do
  - When an operand is read place it immediately in the output.
  - When an operator is read,
    - If (TopOfStack has precedence $\geq$ operator) then Pop operators off the stack until TopOfStack has lower precedence
    - Stack current Operator.
  - When a left bracket is read place in on the stack.
  - When a right bracket is read pop all elements until left bracket. Pop left bracket.

  endwhile
- Pop all elements off the stack.

Stacks III

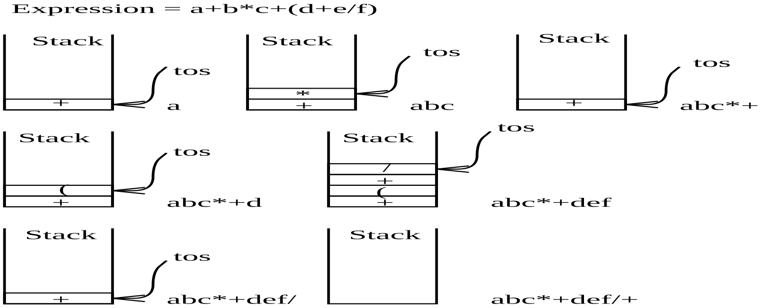


Figure: Conversion from infix to postfix

Implementation of Stacks I

```
/* begin {Definition of class stack} */
#define STACK_SIZE 256
class Stack {
  char string[STACK_SIZE];          // A Stack of characters
  int tos;                          // A pointer to the top o
public:
  void makeNull();                  // creates an Empty Stack
  void push(char x);                // pushes an element on to
  char top();                       // returns the element on
  char pop();                       // Pops an element from th
  int empty();                      // returns 1 if Stack is
 };
/* end{Definition of class Stack} */
```

Implementation of Stacks II

```
/* begin{Implementation of the class Stack} */
void Stack::makeNull() {
  tos = STACK_SIZE;
}
void Stack::push(char x) {
  tos--;
  string[tos] = x;
}


char Stack::top() {
  if (tos < STACK_SIZE)
    return string[tos];
  else
    return(0);
  }
char Stack::pop() {
  char            tmp;
  if (tos >= STACK_SIZE) return 0; else {
```

Implementation of Stacks III

```
  tmp = string[tos];
  tos++;
  return tmp;
  }
}
int Stack::empty() {
  if (tos >= STACK_SIZE)
          return 1;
  else
          return 0;
}
/*end {Implementation of class Stack} */
```

Time complexity of Stack operations using arrays - $O(1)$.

Assignment:

1. Modify the infix to postfix converter to work for right associative operators.
2. Write a function that uses a stack, to determine whether the parentheses are balanced in C program. The parentheses to be considered are $\{, [, (,),], \}$.
3. Write a function that uses a stack, to evaluate a postfix expresssion.

Queues I

Definition: A special kind of ADT, where items are inserted at one end (called the *rear*) and deleted at the other end (called the *front*) - first-in-first-out (FIFO).

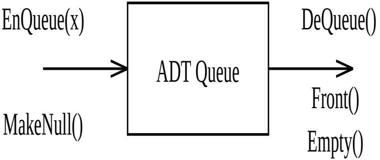


Figure: The ADT Queue

Queues II

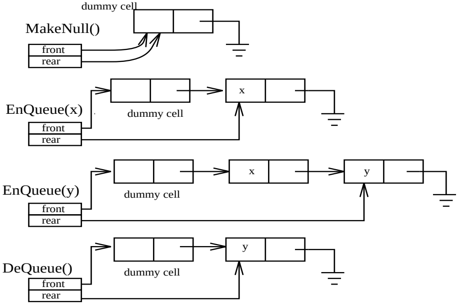


Figure: A sequence of Queue Operations

Implementation of Queues I

/* Definition of an element of the Queue */

```
typedef struct CellType* Position;
struct CellType {
  char subString[10];
  Position next;
};

/*Definiton of class Queue */

class Queue { /* begin {Definition of class Queue} */
private:
  Position front,rear;
public:
  void makeNull();                            // create a new
  void enQueue(char* x);                    // insert x into Qu
  char* front();                            // get element at th
  char* deQueue();                            //delete element 
```

Implementation of Queues II

```
  int empty();                                    // check whether
}; /* end {Definition of class Queue} */

/* begin {Implementation of class Queue} */

void Queue::enQueue(char* x) {
  rear->next = new CellType;
  rear = rear->next;
  strcpy(rear->subString,x);
  cout << rear->subString;
  rear->next = NULL;
}

char* Queue::deQueue() {
char*                temp;
   if (empty())
     cout << "Queue is empty \n";
   else {
```

Implementation of Queues III

```
        temp = front->next->subString;
        front = front->next;
        return(temp);
    }
}
```

Time complexity of queue operations using Linked list - $O(1)$.

Continued.

Issues in implementation of Queues using arrays:

1. Linear Queue exhausts array size.
2. Implement as a circular queue.
   1. Difficult to distinguish between empty and full queue.
   2. Needs an additional variable.

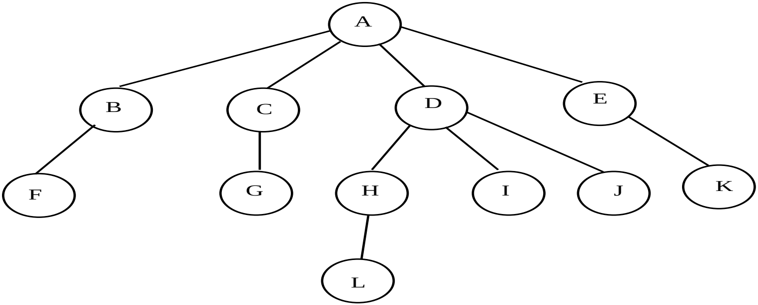
Assignment:

1. Using the Queue ADT simulate a petrol pump.
2. Write a function to store infinitely long strings in a queue.
3. Implement a Queue ADT using a circularly linked list such that all operations on the Queue are performed in $O(1)$ time.

Trees I

Definition: A tree is a hierarchical data structure with a finite set of one or more nodes such that:

- There is a specially designated node called the *root*
- The remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, T_2, ..., T_n$, where each set is a tree. $T_1, T_2, ..., T_n$ are called subtrees of the root.



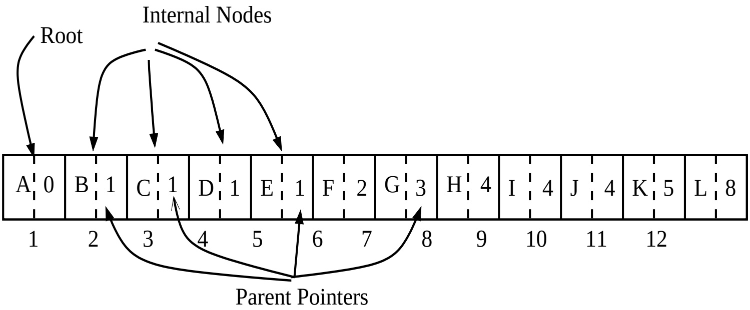
Some terminology with respect trees

Trees II

1. Every tree has a designated noted called the root.
2. Access to any element in the tree is via the root.
3. There are two types of nodes in a tree:
    1. Internal nodes – root and all other nodes that have subtrees
    2. Leaves – are the nodes at the bottom of a tree – they have no subtrees.
4. Height of tree: The length of the longest path from the root to a leaf node is called the height of a tree.
5. Level of a node in a tree: number of levels from root to a given node. Root node is at level 1.

Representation of trees: Arrays I

Array Representation



Every node has a pointer to its parent. This representation is useful for representing sets, memberships. This also useful for languages that donot support recursive data structures.

Representation of trees: Linked lists I

Any general tree can have a variable number of subtrees. A general tree is represented by a set of nodes, each node has two pointers

- A pointer to its left most child
- A pointer to its right sibling

The root has not right sibling

Representation of trees: Linked lists II

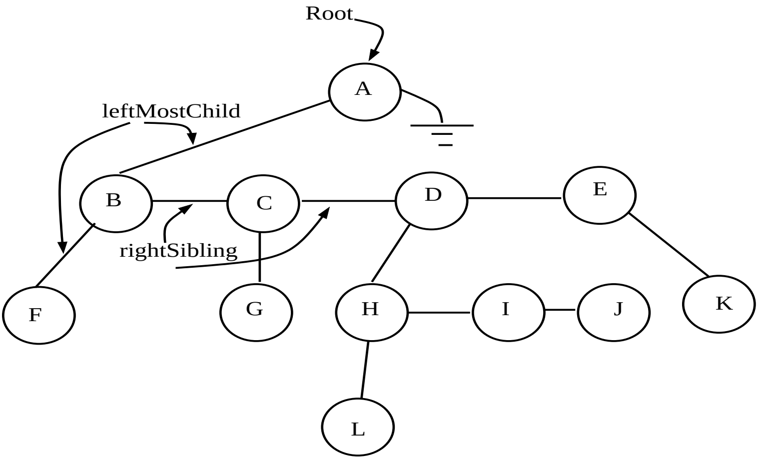


Figure: Representation of trees using leftMostChild, rightSibling representation

Binary Trees I

Definition: A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the left and right subtrees.

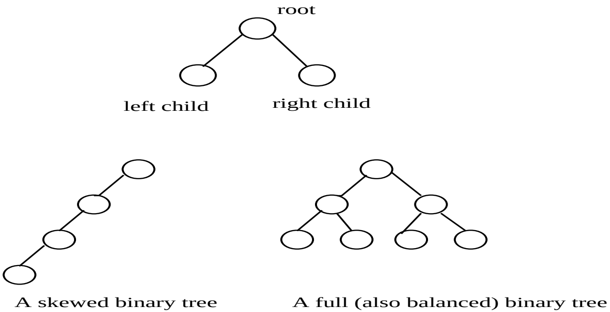


Figure: Binary trees

Operations on binary trees: I

- CreateBinTree(v,lChild,rChild) - Create a binary tree with lChild and rChild as children and v as root.
- makeNull() - Create an empty tree.
- Parent(n) - returns the parent of a node n.
- leftChild(n) - returns the leftChild a node n.
- rightChild(n) - returns the rightChild of a node n.
- root - returns the root of tree

Applications of Binary Trees: I

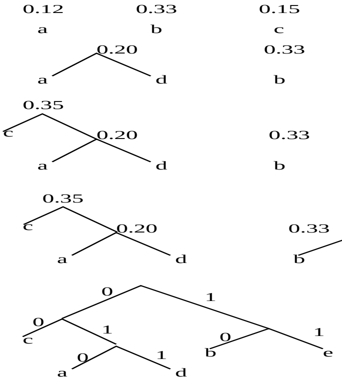
- Representation of Sets
- **Huffmann coding**
- Heaps
- Dictionaries

Huffmann coding: Encodes a message consisting of a sequence of characters. In each message, the **characters** are **independent** and appear with a known **probability** of occurence in any **position**, the probability being th **same** for **all positions**. The encoding should be such that no code for a character is a **prefix** for any other character. Also, the **shortest average codelength** must be obtained.

Applications of Binary Trees: II

Forest

wt	root
0.12	a
0.33	b
0.15	c
0.08	d
0.32	e



a - 010, b - 10, c - 00,
d - 011, e - 11

Figure: Huffmann tree Construction

Huffmann tree construction:
Initialisation: Assume that every character is in a binary tree by itself.
We therefore have as many binary trees as there are letters in alphabet.
While there is more than one tree in the forest do

- get the tree with the smallest weight in the forest, say lC.
- get the tree with the next smallest weight in the forest, say rC.
- create a new node with leftChild as lC and rightChild as rC.
- replace the lC tree in the forest by the tree whose root is the new node, with weight equal to the sum of weights of lC and rC.
- delete tree rC from the forest.

endwhile

Implementation of Binary Trees I

```
/* Definition of a node for a Huffmann tree */
typedef struct tnode *treePtr;
typedef struct tnode {
    treePtr left;
     float probability ;
    treePtr right;
} Treenode;

/* begin {class Huffmann Tree} */
class HuffTree {
  treePtr root;
 public:
 /*Create a tree with a single node */
 void createTree(float prob){
   root = new Treenode;
   root->probability = prob;
   root->left = NULL;
```

Implementation of Binary Trees II

```
  root->right = NULL;
}
 /* Make a single tree from two trees */
 void makeBinTree(HuffTree ltree, HuffTree rtree) {
   float value;
   treePtr lsubtree, rsubtree;
   value = ltree.prob() + rtree.prob();
   root = new Treenode;
   root->probability = value;
   lsubtree = ltree.getRoot();
   rsubtree = rtree.getRoot();
   root->left = lsubtree;
   root->right = rsubtree;
 }

 /* Get the root of the tree */
 treePtr getRoot() {
   return(root);
```

Implementation of Binary Trees III

```
}
  /* Return the probability of the root */
  float prob () {
    return(root->probability);
  }
  /* Print the tree */
  void printTree (treePtr temp) {
    if (temp != NULL) {
      cout << temp->probability << " ";
      printTree(temp->left);
      printTree(temp->right);
    }
  }
```

Traversal of Trees I

- Preorder
- Inorder
- Postorder

Preorder Traversal:

- Visit root
- Visit left child in Preorder recursively
- Visit right child in Preorder recursively

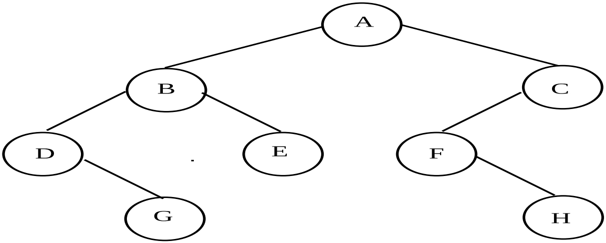
Inorder Traversal:

- Visit left child in Inorder recursively
- Visit root
- Visit right child in Inorder recursively

Postorder Traversal:

- Visit left child in Postorder recursively
- Visit right child in Postorder recursively
- Visit root

Construction of Unique Binary tree from results of traversal



Preorder traversal: ABDGECFH

Inorder traversal: DGBEAFHC

Figure: Binary tree construction from traversals

Binary Tree construction

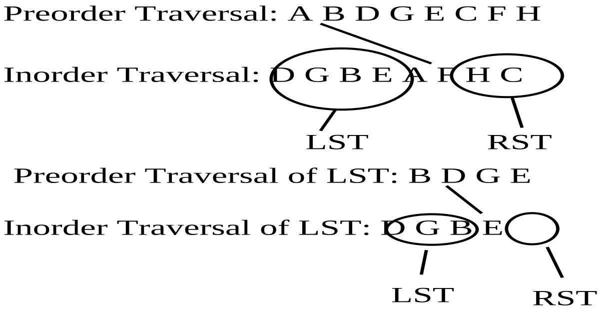


Figure: Binary tree construction from traversals

Binary Search Trees

Definition: A binary search tree is a binary tree. It may be empty. If it is not empty, it satisfies the following properties:

- Every element has a **key** and no two elements have the same key, that is, the **keys are unique**.
- The keys in a nonempty left subtree must be **smaller** than the key on the **root**.
- The keys in a nonempty right subtree must be **larger** than the key at the **root**.
- The **left** and **right** subtrees are also **binary search trees**.

Binary Search Trees are used for Searching (recursively) - similar to binary search on an array of n elements.
Time Complexity of Searching: $O(h+1)$ where h is the height of the tree.

Continued.

Binary Search Trees are identical to Binary Trees except that the property at every node must be maintained.

Both Insertion and Deletion into a Binary Search Tree can cause the tree to become skewed.

Insertion:

Insertion is always done at a leaf node.

Deletion:

When a node is deleted, it is replaced by the **rightmost** child of its **left child**, or the **leftmost** child of **right child**.

Insertion into a Binary Search Tree

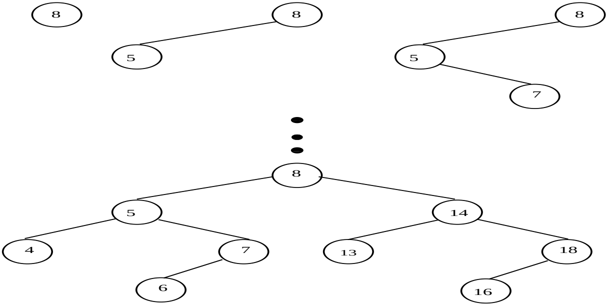


Figure: Insertion into a Binary Search Tree

Deletion from a Binary Search Tree

Suppose we want to delete the node that is marked as 14.

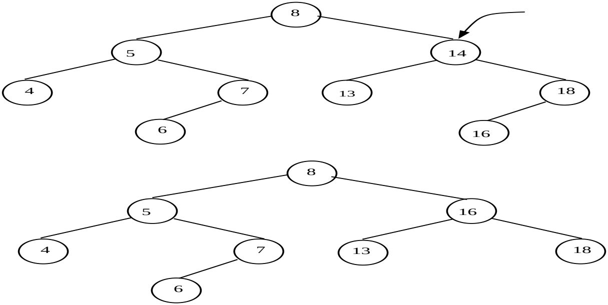


Figure: Deletion from a Binary Search Tree

Other problems on binary trees I

- Maximum number of nodes in a binary tree of height h $2^{h+1} - 1$. Prove using induction.
- Relation between the number of leaf nodes and internal nodes of degree 2 (n_2) in a binary tree: $n_0 = n_2 + 1$. n_0 is the number of nodes of degree 0.
  - Let n be the number of nodes in a binary tree.
  - $n = n_0 + n_1 + n_2$, where n_0, n_1 and n_2 correspond to nodes with degree 0, 1, 2, respectively.
  - If B is the number of branches in a tree, then $n = B + 1$, since every node except the root have a branch leading into it.
  - But $B = n_1 + 2n_2$.
  - Therefore $n_0 = n_2 + 1$.
- How many different binary trees can be formed with 3 unlabeled nodes?
- How many different binary search trees can be formed with 3 labeled nodes?
- Write a simple function to create a mirror image of a binary search tree.

Balanced Binary Trees

To ensure that the height of the tree is of the order $log_2 n$, a BST has to be balanced.

Balanced Binary Trees

- AVL trees - uses special rotations after every insertion and deletion
- Splay trees - uses the move to root heuristic when an element is accessed.

Implementation of Insertion in a Binary Search Tree

```
treeptr *insert(treeptr tree, int number) {
  if (tree == NULL) {
    tree = new tree;
    tree->symbol = number;
    tree->lChild = NULL;
    tree->rChild = NULL;
  } else if (number < tree->symbol)
    tree->lChild = (treeptr) insert(tree->lChild, number)
  else if (number > tree->symbol)
    tree->rChild = (treeptr) insert(tree->rChild, number)
  return(tree);
}
```

Priority Queues - Heaps I

Definition: A max tree is a tree in which the key value in each node is no smaller than the key values in its children, if any. A **max heap** is a complete binary tree that is also a **max tree**. Operations on Heaps:

- Creation of an empty Heap
- Insertion of a new element into a Heap
- Deletion of the largest element from the Heap

Priority Queues are implemented using Heaps.
Time Complexity: $O(log_2 n)$ for both insertion and deletion
Representation of Priority Queues using either linear linked list or linear array is not efficient: Either deletion or insertion takes $O(n)$ time.
Priority Queues are implemented as Complete Binary trees using Arrays.

Continued.

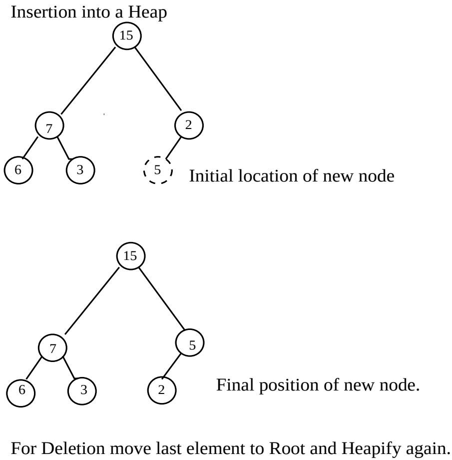


Figure: Heaps - Insertion

Implementation of Heaps I

```
/*  The class Heap defines an ADT Heap. */
/* begin {Definition of class Heap} */
#define HEAP_SIZE 256
typedef int ElemType;
typedef int Position;
class Heap {
  ElemType priority[HEAP_SIZE];          // A Heap of integer
  Position last;
public:
  void insertHeap(ElemType x);            // Insertes an eleme
  ElemType deleteRoot();                  // DeleteRoot of the
  int empty();                            // returns 1 if Heap
  void createHeap();                      // creates an empty
  void printHeap();                       // Prints the heap
 };
/* end{Definition of class Heap} */
```

Implementation of Heaps II

```
/* begin{Implementation of the class Heap} */
void Heap::insertHeap(ElemType x) {
  int                    i;

  last = last + 1;
  i = last;
  while (( i != 1) && (x > priority[i/2])) {
    priority[i] = priority[i/2];
    i = i/2;
  }
  priority[i] = x;
}
ElemType Heap::deleteRoot() {
  ElemType               item,temp;
  Position               parent, child;
  int                    flag;
  item = priority[1];
  parent = 1;
```

Implementation of Heaps III

```
  child = 2;
  temp = priority[last];
  last = last - 1;
  flag = 0;
  while ((child <= last) && (!flag)){
    if ((child < last) && (priority[child] < priority[child
      child = child+1;
    if (temp >= priority[child])
      flag = 1;
    else {
      priority[parent] = priority[child];
      parent = child;
      child = parent*2;
    }
  }
  priority[parent] = temp;
  return(item);
}
```

Implementation of Heaps IV

```
void Heap::createHeap() {
    last = 0;
/* end {Implementation of Heaps} */
```

The Symbol Table ADT I

- Determine if a particular name is in a table
- Retrieve the attributes of that particular name
- Modify the attributes of that name
- Insert a new name and its attributes
- Delete a name and its attributes

The Symbol Table ADT II

Can we use any of the following data structures?

- Use Lists?
- Use Queues?
- Use Stacks?
- Use Trees?

Issues: When these data structures are used, the search times can be large depending upon the location of the item that is being searchedmodified.

Can we do better?

Hash Table I

Key Idea: Use of a *hash function* to access an element.

- Static Hashing
- Dynamic Hashing

Static Hashing:

- Identifiers are stored in a fixed-size table called the **hash table**.
- The address or location of an identifier, x, is obtained by computing some arithmetic function, h, of x.
- $h(x)$ gives the address of x in the table.
- The **hash table**, say ht is partitioned into b buckets, $ht[0], ht[1], ..., ht[b-1]$.
- Each bucket is capable of holding s records.
- $h(x)$ maps the set of possible into the integers 0 through $b-1$.

Hash Table II

Why is hashing effective?
Example: An identifier 6 characters long $\implies$, that there are

$$T = \sum_{0 \leq i \leq 5} 26 * 36^i$$

distinct possible values for x, but any application only uses a small fraction of these.

Hash Table III

Definitions:

- Identifier Density: n/T, where n is the number of identifiers and T is total number of possible identifiers.
- Loading factor: $\alpha = n/(sb)$.
- Synonyms: $h(I_1) = h(I_2)$.
- Overflow: A new identifier I is hashed into a full bucket.
- collision: When two nonidentical identifiers hash to the same bucket.

An Example

0	A	A2
1		
2		
3	D	
4		
5		
6	GA	G
7		
25		

Hash Table with 26 buckets and two slots per bucket

Figure: Hashing based on the letters of the alphabet

Hash Functions

A *hash function*, h, transforms an identifier, x, into its bucket address.

- Mid-Square - Square the identifier and then use an appropriate number of bits from the middle to obtain the bucket address.
- Division - Divide identifier, i.e. compute the remainder when x is divided by M, $x\%M$, and use this as the hash address.
- Folding - Partition identifier into different parts, add partitions and use this as the address.