Advanced Sets (I)
Binary Search Trees

- Binary search trees
  - Represent sets whose elements are linearly ordered

Operations

- INSERT
- DELETE
- MEMBER
- MIN

- $O(\log n)$ steps on the average for a set of $n$ elements
Binary Search Trees

- Binary trees
  - Two or one child nodes for non-leaf nodes

- Binary search trees
  - Binary trees whose all nodes satisfy the following property
    - all elements stored in the left subtree of node $x$ are all less than the element at node $x$, and
    - all elements stored in the right subtree of node $x$ are all greater than the element at node $x$
Binary Search Trees - Examples
Binary Search Trees

- Each node has
  - Element and two subtrees

```plaintext
type
  nodetype = record
    element: elementtype;
    leftchild, rightchild: ^nodetype
  end;
```

- BST set
  - Pointer to the root node of binary search tree

```plaintext
type
  SET = ^nodetype;
```
function MEMBER (x: elementtype; A: SET): boolean,
{ returns true if x is in A, false otherwise }
begin
  if A = nil then
    return (false) { x is never in Ø }
  else if x = A ↑.element then
    return (true)
  else if x < A ↑.element then
    return (MEMBER(x, A ↑.leftchild ))
  else { x > A ↑.element }
    return (MEMBER(x, A ↑.rightchild ))
end; { MEMBER }
Binary Search Trees - Operations

- INSERT

```plaintext
procedure INSERT ( x: elementtype; var A: SET );
{ add x to set A }
begin
  if A = nil then begin
    new (A);
    A.element := x;
    A.leftchild := nil;
    A.rightchild := nil
  end
  else if x < A.element then
    INSERT (x, A.leftchild)
  else if x > A.element then
    INSERT (x, A.rightchild)
  { if x = A.element, we do nothing: x is already in the set }
end; { INSERT }
```
Binary Search Trees - Operations

- DELETEMIN

```plaintext
function DELETEMIN( var A: SET ): elementtype;
{ returns and removes the smallest element from set A }
begin
  if A↑.leftchild = nil then begin
    { A points to the smallest element }
    DELETEMIN := A↑.element;
    A := A↑.rightchild
    { replace the node pointed to by A by its right child }
  end
  else { the node pointed to by A has a left child }
    DELETEMIN := DELETEMIN(A↑.leftchild)
end; { DELETEMIN }
```
Binary Search Trees - Operations

**DELETE**

```plaintext
procedure DELETE ( x: elementtype; var A: SET );
    { remove x from set A }
begin
    if A <> nil then
        if x < A ↑.element
            DELETE(x, A ↑.leftchild)
        else if x > A ↑.element then
            DELETE(x, A ↑.rightchild)
            { if we reach here, x is at the node pointed to by A }
        else if (A ↑.leftchild = nil) and (A ↑.rightchild = nil) then
            A := nil  { delete the leaf holding x }
        else if A ↑.leftchild = nil then
            A := A ↑.rightchild
        else if A ↑.rightchild = nil then
            A := A ↑.leftchild
        else  { both children are present }
            A ↑.element := DELETEmIN(A ↑.rightchild)
    end;  { DELETE }
```
Binary Search Tree vs. Hash Table

- **Binary search tree**
  - INSERT, DELETE, MEMBER, DELETEMIN
  - $\log n$ steps on the average

- **Hash table**
  - Operations except MIN take constant time
  - MIN takes $O(n)$ for n element set
  - If MIN is not needed (or not frequently used), hash table is better
Binary Search Tree vs. Partially Ordered Tree

- **Binary search tree**
  - INSERT, DELETE, MEMBER, DELETEMIN
  - $\log n$ steps on the average

- **Partially ordered tree**
  - Operations take $O(\log n)$ not only on the average, but also in the worst case – INSERT, DELETEMIN
  - But no general DELETE is supported
  - MEMBER requires $O(n)$
Tries

- Special structure for sets of character strings
  - Generally strings of any object types, e.g. integers

- Operations
  - INSERT
  - DELETE
  - MAKENULL
  - PRINT
Tries

- Characteristics of tries
  - Share the same prefix among multiple words
  - Each path from the root to a leaf corresponds to one word
  - *Endmarker* symbol, $, to the ends of all words
  - To avoid confusion between words THE and THEN
  - No prefix of a word can be a word itself
    - Assume all words are $ terminated
Tries – a trie example

\{ \text{THE, THEN, THIN, THIS, TIN, SIN, SING} \}
Tries – trie nodes

- Characteristics of trie nodes
  - Each node has at most 27 children, A, B, …, Z, $
  - Most node will have many fewer than 27 children
  - A leaf reached by an edge labeled $ cannot have any children, and may not be there

- Trie node
  - Mapping : { A, B, …, Z, $ } → “pointer to trie node”
  - Mapping to the next level trie nodes
  - Mapping of $ can be either nil or a self-pointer
### Tries – trie node

- **Abstract data type TRIENODE**
  - ASSIGN \((node, c, p)\) \(node\): current trie node
  - VALUEOF \((node, c)\) \(c\): character
  - GETNEW \((node, c)\) \(p\): pointer to next level trie node

- **Implementation of TRIENODE**
  - Array – mapping for 27 characters, A, B, …, Z, $
  - List – add mapping when needed
Tries – trie nodes (array implementation)

type
cchars = ('A', 'B', ..., 'Z', '$');
TRIENODE = array[cchars] of ↑TRIENODE;
TRIE = ↑TRIENODE;

procedure ASSIGN ( var node: TRIENODE; c: cchars; p: ↑TRIENODE );
begin
  node [c] := p
end; { ASSIGN }

function VALUEOF ( var node: TRIENODE; c: cchars ) ↑TRIENODE;
begin
  return (node [c])
end; { VALUEOF }

procedure GETNEW ( var node: TRIENODE; c: cchars );
begin
  new (node [c]);
  MAKENULL(node [c])
end; { GETNEW }
Tries – operations (array implementation)

procedure INSERT (x: wordtype; var words: TRIE);

var
    i: integer; { counts positions in word x }
    t: TRIE; { used to point to trie nodes corresponding to prefixes of x }

begin
    i := 1;
    t := words;
    while x[i] <> 'S' do begin
        if VALUEOF(t↑, x[i]) = nil then
            { if current node has no child for character x[i], create one }
            GETNEW(t↑, x[i]);
        t := VALUEOF(t↑, x[i]);
        { proceed to the child of t for character x[i], whether or not that child was just created }
        i := i+1 { move along the word x }
    end;
    { now we have reached the first '$' in x }
    ASSIGN(t↑, '$', t)
    { make loop for 'S' to represent a leaf }
end; {INSERT}
Tries – trie node (list implementation)

- Array implementation of trie nodes
  - Takes a collection of words, $p$ different prefixes
  - $27p$ bytes of storage

- List implementation
  - Small domain, 27 characters
  - Mappings defined for few member of that domain
  - Linked list of the characters for which associated value is not the nil pointer