<u>Symbol Tables and Binary Search Trees</u>

Search

- Basic operation for retrieval of a specific piece of information from large volume of previously stored data
- Each data item divided into two parts
  1. Key – used for searching
  2. Record – information to be looked for based on key

  Definition 1 *A **symbol table** is a data structure of items with keys that supports two basic operations:*

  1. *insert a new item, and*
  2. *return an item with a given key.*

  – Also known as a *dictionary*
  – Mostly used to organize software on computers, such as list of variable names in a program during compilation
  – Low-level abstraction or *associative memory*

Symbol Table ADT

- Operations of interest
  1. `insert` a new item
  2. `search` for an item on the basis of a key
  3. `remove` a specified item
  4. `select` the k*th* largest item
  5. `sort` the symbol table
  6. `join` two symbol tables
- Implementation of symbol table ADT

```
class sym_tab
{
    int    num_elements;           // Number of elements in the symbol table
    item * a;                      // Array of items

    // Private functions

    void   sort ( void );
    void   join ( const sym_tab& );

public:
    sym_tab ( void );              // Default constructor
    sym_tab ( const int );         // Parameterized constructor
    sym_tab ( const sym_tab& );    // Copy constructor
    ~sym_tab ( void );             // Destructor

    int    count ( void ) const;   // Number of elements in symbol table
    item&  search ( const key ) const;
```

```
        void   insert ( const item );
        void   remove ( const item );
        item&  select ( const int );
        void   show ( ostream& );
    };
```

- Check the man page for `bsearch(3)` and other searches mentioned in the cross reference section of this man page

Key-indexed search

- Useful when the keys are small compared to the entire record
- The items can be stored in an array, indexed by keys
  - Initialize all items in array `a` to be `NULL`
  - Store the item with key k in location `a[k]`
- Search is straightforward by simply picking the item in `a[k]`
- Deletion is performed by putting a `NULL` item in `a[k]`

Sequential search

Binary search

Binary search trees

- Represented as a linked data structure
- Each node represents an object
- Node contains key + pointer to left child, right child, parent
- Binary-search-tree property
  - All records with smaller keys than a node are in left subtree
  - All records with larger keys than a node are in right subtree
- All keys can be printed in sorted order by *in-order traversal*
- Querying a binary search tree
  - Searching
    * ```
      tree_search (x,k)
         if x = nil or k = key[x] then
            return (x)
         if k < key[x] then
            return (tree_search (left[x],k)
         else
            return (tree_search (right[x],k)
      ```
    * Run-time for `tree_search` is $O(h)$ where h is the height of the tree
  - Minimum and Maximum
    * ```
      tree_minimum (x)
         while left[x] ≠ nil do
            x ← left[x]
         return(x)
      ```

  * `tree_maximum (x)`
```
while right[x] ≠ nil do
    x ← right[x]
return(x)
```
  * Both the procedure run in $O(h)$ time for a tree of height h

- Successor and Predecessor
  * Successor in sorted order determined by *in-order* traversal
  * Successor of node `x` is the smallest key greater than `key[x]`
  * `tree_successor (x)`
```
if right[x] ≠ nil then
    return tree_minimum(right[x])
y ← parent[x]
while y ≠ nil and x = right[y] do
    x ← y
    y ← parent[y]
return y
```

- Insertion and deletion
  - Insertion
    * `tree_insert (T,z)`
```
y ← nil
x ← root[T]
while x ≠ nil do
    y ← x
    if key[z] < key[x] then
        x ← left[x]
    else
        x ← right[x]
parent[z] ← y
if y = nil then
    root[T] ← z
else
    if key[z] < key[y] then
        left[y] ← z
    else
        right[y] ← z
```
    * `tree_insert` runs in $O(h)$ time for a tree of height h
  - Deletion
    * `tree_delete (T,z)`
```
if left[z] = nil or right[z] = nil then
    y ← z
else
    y ← tree_successor(z)
if left[y] ≠ nil then
    x ← left[y]
else
    x ← right[y]
if x ≠ nil
    parent[x] ← parent[y]
if parent[y] = nil then
```

```
      root[T] ← x
   else
      if y = left[parent[y]] then
         left[parent[y]] ← x
      else
         right[parent[y]] ← x
   if y ≠ z then
      key[z] ← key[y]
   return(y)
```

* The procedure runs in $O(h)$ time for a tree of height h

Performance characteristics of BSTs

Index implementations with symbol tables

Insertion at the root in BSTs

BST implementations of other ADT functions