Symbol Tables and Binary Search Trees

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 kth largest item
  5. sort the symbol table
  6. join two symbol tables
- Implementation of symbol table ADT

```cpp
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;
};
```
Symbol Tables and Binary Search Trees

```c
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
    ```c
    * 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
    ```c
    * tree_minimum (x)
      while left[x] ≠ nil do
        x ← left[x]
      return(x)
    ```
Symbol Tables and Binary Search Trees

* tree maximum \( (x) \)
  
  while right[\( x \)] \( \neq \) nil do
  
  \( x \leftarrow \) 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 \( \text{key}[x] \)

* tree successor \( (x) \)
  
  if right[\( x \)] \( \neq \) nil then
  
  return tree minimum(right[\( x \)])

  \( y \leftarrow \) parent[\( x \)]
  
  while \( y \neq \) nil and \( x = \) right[\( y \)] do
  
  \( x \leftarrow y \)

  \( y \leftarrow \) parent[\( y \)]
  
  return \( y \)

• Insertion and deletion

  Insertion

* tree insert \( (T,z) \)

  \( y \leftarrow \) nil
  
  \( x \leftarrow \) root[\( T \)]

  while \( x \neq \) nil do

  \( y \leftarrow x \)

  if key[\( z \)] < key[\( x \)] then

  \( x \leftarrow \) left[\( x \)]

  else

  \( x \leftarrow \) right[\( x \)]

  parent[\( z \)] \( \leftarrow y \)

  if \( y = \) nil then

  root[\( T \)] \( \leftarrow z \)

  else

  if key[\( z \)] < key[\( y \)] then

  left[\( y \)] \( \leftarrow z \)

  else

  right[\( y \)] \( \leftarrow z \)

  return \( z \)

* 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

  \( y \leftarrow z \)

  else

  \( y \leftarrow \) tree successor(\( z \))

  if left[\( y \)] \( \neq \) nil then

  \( x \leftarrow \) left[\( y \)]

  else

  \( x \leftarrow \) right[\( y \)]

  if \( x \neq \) nil

  parent[\( x \)] \( \leftarrow \) parent[\( y \)]

  if parent[\( y \)] = nil then
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