Symbol Tables

Computing 2 COMP1927 16x1
Symbol Tables

- **Searching**: like sorting, searching is a fundamental element of many computational tasks
  - data bases
  - dictionaries
  - compiler symbol tables

- **Symbol table**: a symbol table is a data structure of items with keys that supports at least two basic operations:
  - insert a new item (key,value)
    - (student id, student data) – in a database
    - (word, meaning) – in a dictionary
  - return an item identified by a given key
ABSTRACTING OVER CONCRETE ITEM AND KEY TYPE

- We abstract over the concrete item type by defining these types and some basic operations on them in a separate header file, Item.h:

```c
typedef int Key;

struct record {
    Key keyval;
    char value[10];
};

typedef struct record *Item;

#define key(A) ((A)->keyval)
#define eq(A,B)  {A == B}
#define less(A,B) {A < B}
#define NULLitem NULL // special value for no item

int ITEMscan (Item *);  // read from stdin
int ITEMshow (Item);    // print to stdout
```
SYMBOL TABLE AS ABSTRACT DATA TYPE

Symbol Table ADT:

typedef struct symbolTable *ST;

// new symbol table
ST STinit (void);

// number of items in the table
int STcount (ST);

// insert an item
void STinsert (ST, Item);

// find item with given key
Item STsearch (ST, Key);

// delete given item
void STdelete (ST, Item);

// find nth item
Item STselect (ST, int);

// visit items in order of their keys
void STsort (ST, void (*visit)(Item));
How do we deal with duplicate keys?

- depends on the application:
  - Do not allow duplicates
    - Insertion of duplicates does nothing – fails silently
    - Insertion of duplicates returns an error
  - store all items with the same key in one entry in the symbol table
  - store duplicates as separate entries in the symbol table

- Our approach will not allow duplicates and ignore attempts to insert them.
A SIMPLE SYMBOL TABLE CLIENT PROGRAM

- We start by writing a simple client program:
  - reads items from stdin
  - insert item if not yet in table
  - print resulting table in order
  - print out the smallest, largest and median values.
Symbol tables can be represented in many ways:
- key-indexed array (max # items, restricted key space)
- key-sorted arrays (max # items, using binary search)
- linked lists (unlimited items, sorted list?)
- binary search trees (unlimited items, traversal orders)

Costs (assuming \( N \) items):

<table>
<thead>
<tr>
<th>Type</th>
<th>Search Cost Min</th>
<th>Max</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Indexed Array</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>Key sorted Array</td>
<td>( O(1) )</td>
<td>( O(\log n) )</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>Linked List</td>
<td>( O(1) )</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td>( O(1) )</td>
<td>( O(n) )</td>
<td>( O(\log n) )</td>
</tr>
</tbody>
</table>
IMPLEMENTATION: KEY INDEXED ARRAY

- Use key to determine index position in the array
  - requires dense keys (i.e., few gaps)
  - keys must be integral (or easy to map to integral value)

Properties:
- insert, search and delete are constant time $O(1)$
- init, select, and sort are linear in table size

| items  | NULLitem | 1, data | NULLitem | 3, data | 4, data | 5, data | NULLitem | 7, data |
**IMPLEMENTATION : BINARY SEARCH TREES**

- **Binary tree:**
  - key (and maybe items) in internal nodes
  - key in a node is
    - larger than any key in its left subtree
    - smaller than any key in its right subtree

- **Properties:**
  - init & count are constant time
  - insert, delete, search & select are logarithmic in the number of stored items in average case, linear in worst case (degenerate tree)
  - sort linear in numbers of stored items
IMPLEMENTATION: BINARY SEARCH TREES

- In our implementation, we use a dummy node to represent empty trees.
- Representation of an empty tree:
  - Previously:
  - New implementation:

- Representation of a tree with a single value node:
  - Previously:
  - New implementation:
**Binary Search Tree: Insertion of New Node**

- Insert item with key ‘3’ into tree:

```
    3
   / \
  5   0
 / \  /  \  /
2   0 0
```

```python
def insert(item, key):
    if key < item:
        if item.left is None:
            item.left = Node(key)
        else:
            insert(item.left, key)
    else:
        if item.right is None:
            item.right = Node(key)
        else:
            insert(item.right, key)
```
Binary Search Tree: Insertion of new node

- Insert item with key ‘3’ into tree:
To save space, all the empty subtrees are actually represented by the same struct:
IMPLEMENTATION: BINARY SEARCH TREES

In our implementation, we use a dummy node to represent empty trees:

```c
struct st {
    link root;
};
typedef struct STnode* link;

struct STnode {
    Item item;
    link left,;
    link right;
    int size; // Size of sub-tree rooted at this node
};

static link emptyTree = NULL; // dummy node representing empty tree
static link newNode(Item item, link l, link r, int size);

ST STinit (void) {
    ST st = malloc(sizeof(struct st));
    if(emptyTree == NULL) // only one actual copy of emptyTree is ever created
        emptyTree = newNode(NULLitem, NULL, NULL, 0);
    st->root = emptyTree;
    return st;
}
```
**IMPLEMENTATION : BINARY SEARCH TREES**

- Implementation of recursive insertion:

```c
link insertR (link currentLink, Item item) {
    Key v = key (item);
    Key currentKey = key (currentLink->item);

    if (currentLink == emptyTree) {
        return newNode(item, emptyTree, emptyTree, 1);
    }

    if (less(v, currentKey)) {
        currentLink->left = insertR (currentLink->left, item);
    } else {
        currentLink->right = insertR (currentLink->right, item);
    }

    (currentLink->size)++;
    return currentLink;
}
```
BST: SELECT

- How can we select the $k$th smallest element of a search tree?
- Can be done quite easily if we store the size of the subtree in each node (start with 0)
  - **Base case 1**: if tree is empty tree
    - search was unsuccessful
  - **Base case 2**: if left subtree has $k$ items
    - return node item
  - **Recursive case 1**: left subtree has $m > k$ items
    - continue search of $k$th item in left subtree
  - **Recursive case 2**: left subtree has $m < k$ items
    - continue search of $(k-m-1)th$ item in right subtree
Select Kth Item

- For a tree with N Nodes, indexes are 0..N-1
IMPLEMENTATION : BINARY SEARCH TREES

Implementation of select

```c
static Item selectR (link currentTree, int k) {
    if (currentTree == emptyTree) {
        return NULLItem;
    }
    if (currentTree->left->size == k) {
        return (currentTree->item);
    }
    if (currentTree->left->size > k) {
        return (selectR (currentTree->left, k));
    }
    return (selectR (currentTree->right, k - 1 - currentTree->left->size));
}

Item STselect (ST s, int k) {
    return (selectR (s->root, k));
}
```
Performance Characteristics of BSTs

- We already discussed the performance of binary search trees:
  - on average,
    - $O(\log n)$ steps to search, insert in a tree with $n$ items
  - worst case (degenerate tree)
    - $O(n)$ steps
Symbol Tables as Indexes

- **Scenario:**
  - large set of items;
  - need efficient access via key
  - but also need sequential access to items
  - items might be stored in very large array or file

- **Solution:**
  - leave items in place
  - use symbol table holding (key, ref) pairs
  - Commonly used as an access mechanism in databases.