Symbol Tables

IE 496 Lecture 13

Reading for This Lecture

- Horowitz and Sahni, Chapter 2

Symbol Tables and Dictionaries

- A *symbol table* is a data structure for storing a list of items, each with a *key* and *satellite* data
- The data structure supports the following operations.
  - Construct a symbol table
  - Search for an item having a specified key
  - Insert an item
  - Remove a specified item
  - Count the number of items
  - Print the list of items
- Symbol tables are also called *dictionaries*.
- Note that the keys may not have an ordering

Additional Operations

- If the items can be ordered, we may support the following additional operations
  - Sort the items.
  - Return the maximum or minimum item.
  - Select the k^{th} item.
  - Return the successor or predecessor.
- We may also want to join two symbol tables into one.
- These operation may or may not be supported by various implementations.

Symbol Tables with Integer Keys

- Consider a table whose keys are small positive integers.
- Assuming no duplicate keys, we can implement such a symbol table using an array.

```
class symbolTable
{ private:
    symbolTable();        \\ Disable the default constructor
    Item** st_;           \\ An array of pointers to the items
    const int maxKey_;    \\ The maximum allowed value of a key
  public:
    symbolTable (const int M); \\ Constructor
    ~symbolTable ();           \\ Destructor
    int getNumItems() const;
    Item* search (const int k) const;
    Item* select (int k) const;
    void insert (Item* it);
    void remove (Item* it);
    void sort (ostream& os);
}
```

Implementation

```
symbolTable::symbolTable (const int M)
{
    maxKey_ = M;
    st_ = new Item* [M];
    for (int i = 0; i < M; i++) { st_[i] = 0; }
}

void symbolTable::insert(Item* it) { st_[it.getKey()] = it; }

void symbolTable::remove(Item* it)
{
    delete st_[it.getKey()];
    st_[it.getKey()] = 0;
}

Item* symbolTable::search(const int k) const { return st_[k]; }
```

Implementation (cont.)

```
Item* select(int k)
{
    for (int i = 0; i < maxKey_; i++)
        if (st_[i])
            if (k-- == 0) return st_[i];
}

Item sort(ostream& os)
{
    for (int i = 0; i < maxKey_; i++)
        if (st_[i])
            os << *st_[i];
}

int getNumItems() const
{
    int j(0);
    for (int i = 0; I < maxKey_; i++) if (st_[i]) j++;
    return j;
}
```

Arbitrary Keys

- Note that with arrays, most operations are constant time.
- What if the keys are not integers or have arbitrary value?
- We could still use an array or a linear linked list to store the items.
- However, some of the operations would become inefficient.
- A *binary search tree* (BST) is a more efficient data structure for implementing symbol tables where the keys are an arbitrary data type.

Binary Search Trees

- In a BST data structure, the keys must have an order.
- As with heaps, a binary search tree is a binary tree with additional structure.
- In a binary tree, the key value of any node is
  - greater than or equal to the key value of all nodes in its *left subtree*;
  - less than or equal to the key value of all nodes in its *right subtree*.
- For now, we will assume that all keys are unique.
- With this simple structure, we can implement all functions efficiently.

Searching in a BST

- *Search* can be implemented recursively in a fashion similar to binary search, starting with the root.
  - If the pointer to the current node is *0*, then return *0*
  - Otherwise, compare the search key to the current node's key, if it exists.
  - If the keys are equal, then return a pointer to the current node.
  - If the search key is smaller, recursively search the left subtree.
  - If the search key is larger, recursively search the right subtree.
- What is the running time of this operation?

Inserting a Node

- The procedure for inserting a node is similar to that for searching.
- As before, we will assume there is no item with an identical key already in the tree.
- We perform an unsuccessful search and insert the node in place of the final null pointer at the end of the search.
- This places it where we would expect to find it.
- The running time is the same as searching.
- Constructing a BST from a given list of elements can be done by iteratively inserting each element.

Finding the Minimum and Maximum

- Finding the *minimum* and *maximum* is a simple procedure.
- The minimum is the leftmost node in the tree.
- The maximum is the rightmost node in the tree.

Sorting

- We can easily read off the items from a BST in sorted order.
- This involves walking the tree in a specified order.
- What is it?

Finding the Predecessor and Successor

- To find the successor of a node *x*, think of an inorder tree walk.
- After visiting a given node, what is the next value to get printed out?
  - If *x* has a right child, then the successor is the node with the minimum key in the right subtree (easy to find).
  - Otherwise, the successor is the lowest ancestor of *x* whose left child is also an ancestor of *x* (why?).
  - Note that if a node has two children, its successor cannot have a left child (why not?).
- Finding the predecessor works the same way.

Deleting a Node

- Deleting a node *z* is more complicated than other operations because the structure must be maintained.
- There are a number of algorithms for doing this.
- The most straightforward implementation considers three cases.
  - If *z* has no children, then simply set the pointer to *z* in the parent to be *0*.
  - If *z* has one child, then replace *z* with its child.
  - If *z* has two children, then delete either the predecessor or the successor and then replace *z* with it.
- Why does this work?

Handling duplicate Keys

- What happens when the tree may contain duplicate keys?
- To make things easier, we can always insert items with duplicate keys in the right subtree.
- To find all items with the same key, search for the first item and then recursively search for the same item in the right subtree.
- Alternatively, we could maintain a linked list of items with the same key at each node in the tree.

Performance of BSTs

- Efficiency of the basic operations depends on the depth of the tree.
- Consider the search operation: what is the best case?
- The best case is to make the same comparisons as in binary search.
- However, this can only happen if the root of each subtree is the median element, i.e., the tree is balanced.
- Fortunately, if keys are added at random, this should be the case ``on average."
- What is the worst case?

Selection

- The *selection problem* is that of finding the k^{th} element in an ordered list.
- We need an additional data member in the node class that tracks the size of the subtree rooted at each node.
- With this additional data member, we can recursively search for the k^{th} element
  - Starting at the root, if the size of the left subtree is k-1, return a pointer to the root.
  - If the size of the left subtree is more than \m{k-1}, recursively search for the k^{th} element of the left subtree.
  - Otherwise, recursively search for the $k\text{-}t\text{-}1^{th}$ element of the right subtree, where t is the size of the left subtree.

Balancing

- To guard against poor performance, we would like to have a scheme for keeping the tree balanced.
- There are many schemes for automatically maintaining balance.
- We describe here a method of *manually* rebalancing the tree.
- The basic operation that we'll need is that of *rotation*.
- Rotating the tree means changing the root from the current root to one of its children.

Rotation

- To change the right child of the current root into the new root.
  - Make the current root the left child of the new root.
  - Make the left child of the new root the right child of the old root.
- Note that we can make any node the root of the BST through a sequence of rotations.
- To partition the list around the k^{th} item, select the

Partitioning and Rebalancing

- To partition the list around the item, select the k^{th} item, select the k^{th} item and rotate it to the root.
- This can be implemented easily in a recursive fashion.
- The left and right subtrees form the desired partition.
- To (re)balance a BST.
  - Partition around the middle node.
  - Recursively balance the left and right subtrees.
- This operation can be called periodically.
- What is the running time of this operation?

Delete

- Using the partition operation, we can implement delete in a slightly different way.
  - Partition the right subtree of the node to be deleted around its smallest element x.
  - Make the root of the left subtree the left child of x.

Root Insertion and Joining

- Often it is useful to be able to insert a node as the root of the BST.
- This can be done easily by inserting it as usual and then rotating it to the root, i.e., partition around it.
- With root insertion, we can define a recursive method to join two BSTs.
  - Insert the root of the first tree as the root of the second.
  - Recursively join the pairs of left and right subtrees.

Randomized BSTs

- We used use randomization to guard against worst case behavior.
- The procedure for randomly inserting into a BST of size n is as follows.
  - With probability $1/(n+1)$, perform root insertion.
  - Otherwise, recursively insert into the right or left subtree, as appropriate, using the same method.
- One can prove mathematically that this is the same as randomly ordering the elements first.
- Hence, this should guard against common worst-case inputs.

Hash Tables

- A *hash table* is another easy and efficient implementation of a symbol table.
- It works with keys that are not ordered, but supports only
  - insert
  - delete
  - search
- It is based on the concept of a *hash function*.
  - Maps each possible element into a specified bucket
  - The number of buckets is much less than the number of possible elements
  - Each bucket can store a limited number of elements

Addressing Using Hashing

- Recall the array-based implementation of a dictionary.
- We allocated one memory location for each possible key.
- Using hashing, we can extend this method to the case where the set U of possible keys is extremely large.
- A hash function h takes a key and converts it into an array index (called the *hash value*).
- With a hash function, we can use a very efficient array-based implementation to store items in the table.
- Note that we can no longer do sorting or selection.

Parameters

- T = total number of possible elements
- b = number of buckets
- n = number of elements in the table
- n/T = element density
- $\alpha = n/b$ = load factor

Hash Functions

- *Collision*: two elements map to the same bucket.
- Choosing a hash function
  - easy to compute
  - minimize collisions
- If $P(f(X) = i) = 1/b$ over all possible elements X, then f is a *uniform hash function*.
- It is not easy to find a good hash function.
  - It depends on the distribution of keys
  - We may not know that ahead of time

Significant Bits

- Two obvious hash functions are to simply consider either the first or last k bits of the key.
- These hash functions are very fast to compute (why?).
- However, they are both notoriously bad hash functions, especially for strings (why?).
- One possible way to do better is to use the bits in the middle, though even this is not ideal.

Simple Hash Function

- Interpret each element of the set as an integer X.
- Take the hash function to be

$$f(X) = X \bmod M.$$

- M is the number of buckets.
- The choice of M is critical.
- M should not be a power of 2 or an even number.
- M should be a prime number with some other nice properties (more on this later).

Overflow Handling

- *Open Addressing*: If the hashed address is already used, find a new one by a simple rule.
  - Bad performance when the hash table fills up.
  - Can end up searching the whole table.
- *Chaining*: Form a linked list of elements with the same hash value.
  - Only compares items with same hash value.
  - Good performance with well-distributed hash values.

Analysis with Chaining

- Insertion is constant time, as long as we don't check for duplication.
- Deletion is also constant time if the lists are doubly linked.
- Searching takes time proportional to the length of the list.
  - Depends on how well the hash function performs and the *load factor*.
  - Both search hits and misses take time O(α).

Related Results

- Under reasonable assumptions on the distribution of keys, we can derive some probabilistic results.
- The probability that a given list has more than $t\alpha$ items on it is less than $(\alpha e / t)\, e^{-\alpha}$.
- In other words, if the load factor is 20, the probability of a list with more than 40 items on it is .0000016.
- The average number of items inserted before the first collision occurs is approximately the square root of M.
- The average number of items to be inserted before every list has at least one item is approximately $M \ln M$.

Table Size with Chaining

- Choosing the size of the table is a perfect example of a *time-space* tradeoff.
- The bigger the table is, the more efficient it will be.
- On the other hand, bigger tables also mean more wasted space.
- When using chaining, we can afford to have a load factor greater than one.
- A load factor as high as 5 or 10 can work well if memory is limited.