Symbol Tables and Binary Search Trees

- symbol table: a data structure that supports insert and get of an item using a key

Symbol-Table Abstract Data Type

- operations:
  - insert
  - search (from key)
  - remove
  - select the k-th largest (smallest) item
  - sort
  - join two symbol tables
- C++ standard has bsearch
- item ADT for symbol table:

```
#include <stdlib.h>
#include <iostream.h>
static int maxKey = 1000;
typedef int Key;
class Item {
private:
    Key keyval;
    float info;
public:
    Item() { keyval = maxKey; }
    Key key() { return keyval; }
    int null() { return keyval == maxKey; }
    void rand() {
        keyval = 1000*::rand()/RAND_MAX;
        info = 1.0*::rand()/RAND_MAX;
    }
    int scan(istream& is = cin) { return (is >> keyval >> info) != 0; }
    void show(ostream& os = cout) { os << keyval << " " << info << endl; }
};

ostream& operator<<(ostream& os, Item& x) {
    x.show(os); return os;
}
```

- also want operators == and <
- symbol table ADT:

```
template<typename Item, typename Key>
class Symbol_table {
```

```
private:
    // Implementation-dependent code
public:
    ST(int);
    int count();
    Item search(Key) ;
    void insert(Item);
    void remove(Item);
    Item select(int);
    void show(ostream&);
};
```

- also need appropriate definitions for Key, key(), null(), operator== and operator<
- sorted and unsorted symbol tables (if a key does not support operator<, select and sort are generally not supported)
- symbol tables with duplicate keys and symbol tables with only unique keys
- for duplicate keys:
    - only distinct keys and each item structure for a key maintains a list of values
    - leave all duplicates in the data structure and return ANY item with key
    - assume each item has a separate unique ID and find the item based on that and the key
- sample client:

```
#include <iostream.h>
#include <stdlib.h>
#include "Item.h"
#include "Symbol_table.h"

int main(int argc, char *argv[]) {
    int N, maxN = atoi(argv[1]), sw = atoi(argv[2]);
    Symbol_table<Item, Key> st(maxN);
    for (N = 0; N < maxN; N++) {
        Item v;
        if (sw) v.rand(); else if (!v.scan()) break;
        if (!(st.search(v.key())).null()) continue;
        st.insert(v);
    }
    st.show(cout);
    cout << "\n";
    cout << N << " keys" << endl;
    cout << st.count() << " distinct keys" << "\n";

    return 0;
 }
```

- additional operations:
    - memoized "finger" search
    - range search
    - distance between keys

    - near-neighbor search

Key-Indexed Search

- using indices of distinct small numbers enables a key-indexed array based symbol table:

```
template <typename Item, typename Key>
class Symbol_table {
private:
    Item nullItem, *st;
    int M;
public:
    Symbol_table(int maxN) {
        M = nullItem.key();
        st = new Item[M];
    }
    int count() {
        int N = 0;
        for (int i = 0; i < M; i++) {
            if (!st[i].null()) {
                N++;
            }
        }
        return N;
    }
    void insert(Item x) { st[x.key()] = x; }
    Item search(Key v) { return st[v]; }
    void remove(Item x) { st[x.key()] = nullItem; }
    Item select(int k) {
        for (int i = 0; i < M; i++) {
            if (!st[i].null()) {
                if (k-- == 0) {
                    return st[i];
                }
            }
        }
        return nullItem;
    }
    void show(ostream& os) {
        for (int i = 0; i < M; i++) {
            if (!st[i].null()) {
                st[i].show(os);
            }
        }
    }
};
```

- direct corollary to key-indexed sort
- method of choice when applicable (possible) due to efficiency
- when only keys, can use a table of bits aka existence table

- properties:
  - for distinct positive integer keys less than M, key-indexed search can be implemented with search, insert, and remove running in constant time and initialize, select, and sort running in time proportional to M
- for duplicate keys, can use linked lists as elements in the array
- count here is lazy; can make it eager by maintaining a counter of the non-empty positions -> prefer the lazy approach if the use of count is rare

Sequential Search

- for keys from too large a range to be used as indices it is possible to still use a similar array based approach by storing the items in order and inserting a new item in sorted order

```
template<typename Item, typename Key>
class Symbol_table {
private:
    Item nullItem, *st;
    int N;
public:
    Symbol_table(int maxN) {
        st = new Item[maxN+1]; N = 0;
    }
    int count() { return N; }
    void insert(Item x) {
        int i = N++;
        Key v = x.key();
        while (i > 0 && v < st[i-1].key()) {
            st[i] = st[i-1];
            i--;
        }
        st[i] = x;
    }
    Item search(Key v) {
        for (int i = 0; i < N; i++) {
            if (!(st[i].key() < v)) {
                break;
            }
        }
        if (v == st[i].key()) {
            return st[i];
        }
        return nullItem;
    }
    Item select(int k) { return st[k]; }
    void show(ostream& os) {
        int i = 0;
        while (i < N) st[i++].show(os);
    }
};
```

- can improve inner loop in search by maintaining a sentinel to remove the overflow check

- here search is fast, but insert is slow
- can instead not maintain the array in sorted order and insert is fast but search and sort are slow
- linked list version (unordered):

```
#include <stdlib.h>
template<typename Item, typename Key>
class Symbol_table {
private:
    Item nullItem;
    struct node {
        Item item;
        node* next;
        node(Item x, node* t) {
            item = x;
            next = t;
        }
    };
    typedef node *link;
    int N;
    link head;
    Item searchR(link t, Key v) {
        if (t == 0) {
            return nullItem;
        }
        if (t->item.key() == v) {
            return t->item;
        }
        return searchR(t->next, v);
    }
public:
    Symbol_table(int maxN) {
        head = 0;
        N = 0;
    }
    int count() { return N; }
    Item search(Key v) { return searchR(head, v); }
    void insert(Item x) {
        head = new node(x, head);
        N++;
    }
};
```

- properties:
  - sequential search in an N item symbol table uses N/2 comparisons on average (array or linked lists, ordered or unordered)
  - uses a constant number of steps for inserts and N comparisons for search misses (always)

    - uses N/2 comparisons for insertions, search hits, and search misses on average

Binary Search

- binary search
    - split input in half, compare to find the portion the key belongs to, repeat
- properties:
    - never uses more than lg(n) + 1
    - (the number of comparisons is equal to the number of bits in the binary representation of N)
    - sorting a table causes a slowdown for insert
- binary search for an array based symbol table

```
private:
    Item searchR(int l, int r, Key v) {
        if (l > r) return nullItem;
        int m = (l+r)/2;
        if (v == st[m].key()) return st[m];
        if (l == r) return nullItem;
        if (v < st[m].key()) {
            return searchR(l, m-1, v);
        } else {
            return searchR(m+1, r, v);
        }
    }
public:
    Item search(Key v) { return searchR(0, N-1, v); }
```

- for binary search with a given input, the sequence is predetermined
- improvements:
    - guess first comparison with more precision (interpolation search) (assumes key value is is numerical and evenly distributed)

```
int m = (l + r) / 2;
// ->
int m = l + (v - a[l].key()) * (r - l) / a[r[.key() - a[l].key());
```

Binary Search Trees (BSTs)

- using a tree overcomes the cost of insertion in a sorted implementation for binary sort
- (this is one of the most fundamental algorithms in computer science)
- definition:
    - binary search tree: a tree with keys associated with each node and the the keys in the left subtree of each node is less than the key at the node and the keys in the right subtree of each node is greater than the key at the node
- binary search tree symbol table

```
template<typename Item, typename Key>
class Symbol_table {
private:
    struct node {
        Item item;
        node *l, *r;
        node(Item x) {
            item = x;
            l = 0;
            r = 0;
        }
    };
    typedef node *link;
    link head;
    Item nullItem;
    Item searchR(link h, Key v) {
        if (h == 0) return nullItem;
        Key t = h->item.key();
        if (v == t) {
            return h->item;
        }
        if (v < t) {
            return searchR(h->l, v);
        } else {
            return searchR(h->r, v);
        }
    }
    void insertR(link& h, Item x) {
        if (h == 0) {
            h = new node(x);
            return;
        }
        if (x.key() < h->item.key()) {
            insertR(h->l, x);
        } else {
            insertR(h->r, x);
        }
    }
public:
    Symbol_table(int maxN) { head = nullptr; }
    Item search(Key v) { return searchR(head, v); }
    void insert(Item x) { insertR(head, x); }
};
```

- can also use dummy nodes
- sorting with a BST:

```
private:
void showR(link h, ostream& os) {
    if (h == nullptr) {
        return;
    }
    showR(h->l, os);
    h->item.show(os);
```

```
    showR(h->r, os);
}
public:
    void show(ostream& os) { showR(head, os); }
```

- non-recursive BST insertion

```
void insert(Item x) {
    Key v = x.key();
    if (head == nullptr) {
        head = new node(x);
        return;
    }
    link p = head;
    for (link q = p; q != 0; p = q ? q : p) {
        q = (v < q->item.key()) ? q->l : q->r;
    }
    if (v < p->item.key()) {
        p->l = new node(x);
    } else {
        p->r = new node(x);
    }
}
```

- the non-recursive BST operations require maintaining a link to the parent of the current node
- duplicate keys can be used in BSTs but will result in scattered positioning throughout the tree

Performance Characteristics of BSTs

- dependent on shapes of trees
- height is worst case costs
- properties:
  - search hits require 2 * ln(N) or about 1.39 * lg(N) on average from N random keys
  - insertions and search misses require about 2 * ln(N) or about 1.39 * lg(N) comparisons on average
  - in the worst case a search in a binary search tree with N keys can require N comparisons
- a recurrence relationship exists that describes the number of comparisons and internal path length

Index Implementations with Symbol Tables

- same idea as sort -> use an external search structure that indexes into another
- can construct a binary search tree using this technique
- use three arrays:
  - one for the items

    - one for left links
    - one for right links

```
x = x->l
// becomes
x = l[x]
```

- allows for the addition of arrays containing extra information about each element to be added without adding tree processing routines
- inappropriate for space restricted environments or trees that grow and shrink in size often
- indexing a string for array indexed based binary search

```
#include <iostream.h>
#include <fstream.h>
#include "Item.h"
#include "Symbol_table.h"
static char text[maxN];
int main(int argc, char *argv[]) {
    int N = 0;
    char t;
    ifstream corpus;
    corpus.open(*++argv);
    while (N < maxN && corpus.get(t)) {
        text[N++] = t;
    }
    text[N] = 0;
    Symbol_table<Item, Key> st(maxN);
    for (int i = 0; i < N; i++) st.insert(&text[i]);
    char query[maxQ]; Item x, v(query);
    while (cin.getline(query, maxQ)) {
        if ((x = st.search(v.key())).null()) {
            cout << "not found: " << query << endl; {
        } else {
            cout <<  x->text << ": " << query << endl;
        }
    }
}
```

Insertion at the root in BSTs

- the standard implementation of BSTs inserts new nodes as external nodes
- another approach is to insert at the root
- rotation (right):
    - get link to new root E from left link of element S
    - set left link of S to right link of E
    - set right link of E to S
    - set link to S from S's parent to point to E
    - moves E and its left subtree up one level and S and its right subtree down one level

```
void rotate_right(link& h) {
    link x = h->l;
    h->l = x->r;
    x->r = h;
    h = x;
}

void rotate_left(link& h) {
    link x = h->r;
    h->r = x->l;
    x->l = h;
    h = x;
}
```

- root insertion in BSTs:

```
private:
    void insert_t(link& h, Item x) {
        if (h == 0) { h = new node(x); return; }
        if (x.key() <  h->item.key()) { insertT(h->l, x); rotR(h); }
        else { insertT(h->r, x); rotL(h); }
    }
public:
    void insert(Item item) { insertT(head, item); }
```

BST Implementations of Other ADT Functions

- select, join, and remove are all important functions to be performed on binary trees
- removal can be problematic
- select on BSTs is analogous to quicksort:

```
private:
    Item select_r(link h, int k) {
        if (h == 0) return null_item;
        int t = (h->l == 0) ? 0: h->l->N;
        if (t > k) return select_r(h->l, k);
        if (t < k) return select_r(h->r, k-t-1);
        return h->item;
    }
public:
    Item select(int k) { return selectR(head, k); }
```

- removal requires finding a node in a subtree and recursively removing the node from that subtree
- to combine two trees with one containing all keys greater than the first, partition to bring smallest element in greater tree to root and then linking the left treet to this root
- parititioning of a BST

```
void partR(link& h, int k) {
    int t = (h->l == 0) ? 0: h->l->N;
    if (t > k ) { partR(h->l, k); rotR(h); }
    if (t < k ) { partR(h->r, k-t-1); rotL(h); }
}
```

- possible approaches to removal:
  - add a parent link to facilitate removal (cumbersome and adds space overhead)
  - search for pointer to node being remove and stop when found (?)
  - use reference or pointer to the pointer to the node as the handle (works in C and C++ but often not elsewhere)
  - lazily mark nodes for removal and batch remove and reconstruct the tree
- removal by removing first node with key v:

```
private:
  link joinLR(link a, link b) {
        if (b == 0) return a;
        partR(b, 0); b->l = a;
        return b;
    }
    void removeR(link& h, Key v) {
        if (h == 0) return;
        Key w = h->item.key();
        if (v < w) removeR(h->l, v);
        if (w < v) removeR(h->r, v);
        if (v == w) {
            link t = h;
            h = joinLR(h->l, h->r); delete t;
        }
    }
public:
    void remove(Item x) { removeR(head, x.key()); }
```

- joining BSTs:

```
private:
    link joinR(link a, link b) {
        if (b == 0) return a;
        if (a == 0) return b;
        insertT(b, a->item);
        b->l = joinR(a->l, b->l);
        b->r = joinR(a->r, b->r);
        delete a;
        return b;
    }
public:
    void join(ST<Item, Key>& b) { head = joinR(head, b.head); }
```

- both remove and join above can lead to poorly balanced trees

- BSTs require a large proportion of their space to store links
- poorly balanced trees lead to slow performance

Balanced Trees

- standard binary trees have bad worst case performance
- like quicksort, standard binary trees are susceptible to a number of cases that can easily occur:
  - sorted input
  - many duplicate keys
  - reverse order
  - alternating large and small keys
  - any large segment with a simple structure
- balanced trees are preferable
- one approach is to periodically rebalance the tree
- balance a binary tree:

```
void balanceR(link& h) {
    if ((h == nullptr) || (h->N == 1)) return;
    partR(h, h->N/2);
    balanceR(h->l);
    balanceR(h->r);
}
```

- three general approaches to providing performance guarantees:
  - randomize
    - introduce random decisions to reduce the probability of worst case performance (randomized BSTs and skip lists)
  - amortize
    - do more work at certain points to provide a guaranteed upper bound on average per-operation cost (splay BSTs)
  - optimize
    - take the time to provide specific performance guarantees for each operation

Randomized BSTs

- assuming items inserted in random order means each node is equally likely to be the root node
- inserting randomness into the algorithm makes this hold without any assumptions
- each new node should be the new root with a probability of 1 / (N + 1), so root insertion is randomly done with this probability

- properties:
  - building a randomized BST is equivalent to building a standard BST from a random initial permutation of the keys
  - 2 * N *lg(N) comparisons for construction and 2 * ln(N) comparisons for search
  - the probability that the construction cost for a randomized BST is more than a factor of a X the average is e ^ -a
  - making a tree with an arbitrary sequence of randomized insert, remove, and join operations is equivalent to building a standard BST from a random permutation of the input keys (see Karp's general solution to probabilistic recurrence relations)
  - the implied cost of searching in a randomized BST is close to the average
- randomized BST insertion:

```
private:
    void insertR(link& h, Item x) {
        if (h == 0) { h = new node(x); return; }
        if (rand() < RAND_MAX/(h->N+1)) {
            insertT(h, x);
            return;
        }
        if (x.key() < h->item.key()) {
            insertR(h->l, x);
        } else {
            insertR(h->r, x);
        }
        h->N++;
    }
public:
    void insert(Item x) { insertR(head, x); }
```

- join randomized BSTs

```
private:
    link joinR(link a, link b) {
        if (a == nullptr) return b;
        if (b == nullptr) return a;
        insertR(b, a->item);
        b->l = joinR(a->l, b->l);
        b->r = joinR(a->r, b->r);
        delete a; fixN(b); return b;
    }
public:
    void join(ST<Item, Key>& b) {
        int N = head->N;
        if (rand()/(RAND_MAX/(N+b.head->N)+1) < N) {
            head = joinR(head, b.head);
        } else {
            head = joinR(b.head, head);
```

```
        }
    }
```

- deletion in a randomized BST

```
link joinLR(link a, link b) {
    if (a == nullptr) return b;
    if (b == nullptr) return a;
    if (rand()/(RAND_MAX/(a->N+b->N)+1) < a->N) {
        a->r = joinLR(a->r, b); return a;
    } else {
        b->l = joinLR(a, b->l); return b;
    }
}
```

- good to understand probability theory to be able to understand these performance properties
- randomized BSTs are one of the easiest ways to create a symbol table with near optimal performance guarantees

Splay BSTs

- splay insertions bring newly inserted nodes to the root (standard root insertion when the links from the root to the grandchild on the search path have the same orientation) and two rotations when the intermediate links have the same orientation
- properties
    - the number of comparisons used when a splay BST is built from N insertions into an initially empty tree is O(N * lg(N))
    - the number of comparisons required for any sequence of M insert or search operation in an n-node splay BST is O((N + M) lg(N + M))
- splay insertion in BSTs

```
private:
    void splay(link& h, Item x) {
        if (h == 0) { h = new node(x, 0, 0, 1); return; }
        if (x.key() < h->item.key()) {
            link& hl = h->l; int N = h->N;
            if (hl == 0) { h = new node(x, 0, h, N+1); return; }
            if (x.key() <  hl->item.key()) { splay(hl->l, x); rotR(h); }
            else { splay(hl->r, x); rotL(hl); }
            rotR(h);
        } else {
            link &hr = h->r; int N = h->N;
            if (hr == 0) { h = new node(x, h, 0, N+1); return; }
            if (hr->item.key() < x.key()) { splay(hr->r, x); rotL(h); }
            else { splay(hr->l, x); rotR(hr); }
            rotL(h);
        }
```

```
    }
public:
    void insert(Item item) { splay(head, item); }
```

- checks the following configurations for rotations:
  - left-left: rotate right at the root twice
  - left-right: rotate left at the left child, then right at the root
  - right-right: rotate left at the root twice
  - right-left: rotate right at the right child, then left at the root

Top_down 2-3-4 Trees - TODO: fill this in more

- definition:
  - a tree that either is empty or comprises three types of nodes:
    - 2 nodes:
      - 1 key
      - left link -> smaller keys
      - right link -> larger keys
    - 3 nodes:
      - 2 keys
      - left link -> smaller keys
      - middle link -> keys between the two keys
      - right link -> greater keys
    - 4 nodes:
      - 3 keys
      - 4 links -> each range subtended by the keys
  - balanced 2-3-4 search tree: all links to empty trees are the same distance from the root
- there are generalized 2-3-4 trees, but the context implies balanced 2-3-4 trees
- insertion:
  - if the key to be inserted terminates at a 2 node, turn it to a 3 node
  - if the key to be inserted terminates at a 3 node, turn it to a 4 node
  - if the key to be inserted terminates at a 4 node, split it into two 2 nodes and pass the middle key up to the parent
- when the root becomes a 4 node, split it into a triangle of 2 nodes
- properties:
  - searches in an n-node 2-3-4 tree visits at most lg(N + 1) nodes

    - insertions into n-node 2-3-4 trees require fewer than lg(N + 1) node splits in the worst case and seem to require less than one node split on the average
- NOTE: go back and review more info on these trees later

Red-Black Trees - TODO: fill this in more

- models 2-3-4 trees using binary trees and encoding extra information to encode red nodes and black nodes
- coloring the links is equivalent to coloring the nodes
- properties:
    - standard search works without modification
    - correspond to 2-3-4 trees and can maintain balance similarly
    - a search in a red-black tree with N nodes requires fewer than 2 * lg(N + 2) comparisons
    - a search in a red-black tree with N nodes built from random keys uses about 1.002 * lg (N) comparisons on average
- insertion in red-black trees

```
private:
    int red(link x) { if (x == 0) return 0; return x->red; }
    void RBinsert(link& h, Item x, int sw) {
        if (h == 0) { h = new node(x); return; }
        if (red(h->l) && red(h->r)) { h->red = 1; h->l->red = 0; h->r->red = 0; }
        if (x.key() < h->item.key()) {
            RBinsert(h->l, x, 0);
            if (red(h) && red(h->l) && sw) rotR(h);
            if (red(h->l) && red(h->l->l)) { rotR(h); h->red = 0; h->r->red = 1; }
        } else {
            RBinsert(h->r, x, 1);
            if (red(h) && red(h->r) && !sw) rotL(h);
            if (red(h->r) && red(h->r->r)) { rotL(h); h->red = 0; h->l->red = 1; }
        }
    }
public:
    void insert(Item x) { RBinsert(head, x, 0); head->red = 0; }
```

- definitions:
    - red-black BST: a BST with a node marked as red or black with no two consecutive red nodes occurring in any path from an external node to the root
    - balanced red-black BST: red-black BST in which all paths from external nodes to the root have the same number of black nodes

Skip Lists - TODO: fill this in more

- two-level linked list - every third node contains a second link that allows faster traversal

- definition:
  - skip list: an ordered linked list where each node contains a variable number of links, where the i-th links in the nodes implement a singly linked list that skips the nodes with fewer than i links
- properties:
  - search and insertion in a randomized skip list with parameter t require about (t * log sub t (N)) / 2 = (t = (2 * lg(t))) * lg(N)
  - skip lists have (t / (t - 1)) * N links on the average
- searching in skip lists:

```
private:
  Item searchR(link t, Key v, int k)
    { if (t == 0) return nullItem;
      if (v == t->item.key()) return t->item;
      link x = t->next[k];
      if ((x == 0) || (v < x->item.key()))
        {
          if (k == 0) return nullItem;
          return searchR(t, v, k-1);
        }
      return searchR(x, v, k);
    }
public:
  Item search(Key v)
    { return searchR(head, v, lgN); }
```

- general skip list data structure:

```
private:
  struct node
    { Item item; node **next; int sz;
      node(Item x, int k)
        { item = x; sz = k; next = new node*[k];
          for (int i = 0; i < k; i++) next[i] = 0; }
    };
  typedef node *link;
  link head;
  Item nullItem;
  int lgN;
public:
  ST(int)
    { head = new node(nullItem, lgNmax); lgN = 0; }
```

- insertion in skip lists:

```
private:
  int randX()
    { int i, j, t = rand();
      for (i = 1, j = 2; i < lgNmax; i++, j += j)
        if (t > RAND_MAX/j) break;
```

```
      if (i > lgN) lgN = i;
      return i;
    }
  void insertR(link t, link x, int k)
    { Key v = x->item.key(); link tk = t->next[k];
      if ((tk == 0) || (v < tk->item.key()))
        {
          if (k < x->sz)
            { x->next[k] = tk; t->next[k] = x; }
          if (k == 0) return;
          insertR(t, x, k-1); return;
        }
      insertR(tk, x, k);
    }
public:
  void insert(Item v)
    { insertR(head, new node(v, randX()), lgN); }
```

- removal in skip lists:

```
private:
  void removeR(link t, Key v, int k)
    { link x = t->next[k];
      if (!(x->item.key() < v))
        {
          if (v == x->item.key())
            { t->next[k] = x->next[k]; }
          if (k == 0) { delete x; return; }
          removeR(t, v, k-1); return;
        }
      removeR(t->next[k], v, k);
    }
public:
  void remove(Item x)
    { removeR(head, x.key(), lgN); }
```

Performance Characteristics - TODO: fill this in more

- randomized BSTs are the simplest to implement
- see the table

Radix Search

- radix search: search methods that examine the keys one small step at a time
- keys may be words or strings and words are viewed as part of a base-R number system
- C-strings can be viewed as variable length numbers and compared via the process of extract the i-th digit
- reasonable worst case performance without dealing with the issues presented by balanced trees

- may make inefficient use of space and performance can suffer if access to the bytes of the keys is not available

Digital Search Trees

- digital search trees (DSTs): similar to binary trees with the branches based on comparisons of the leading bits in the keys
- make the decision whether to move left or right from the root based on 0 or 1 -> 0 for left 1 for right
- the first key inserted becomes the root
- binary digital search tree:

```
private:
    Item searchR(link h, Key v, int d) {
        if (h == 0) return nullItem;
        if (v ==  h->item.key()) return h->item;
        if (digit(v, d) == 0)
            return searchR(h->l, v, d+1);
        else return searchR(h->r, v, d+1);
    }
public:
    Item search(Key v) { return searchR(head, v, 0); }
```

- properties:
  - a search or insertion in a digital search tree requires about lg(N) comparisons on average and about 2 * lg(N) comparisons in the worst case (for N random keys)
  - the number of comparisons is never more than the number of bits in the search key

Tries

- definition:
  - trie: a binary tree that has keys only associated with leaf nodes. The trie for an empty set of keys is a null link, a single key is a leaf containing that key, and a set of keys of cardinality greater than one is an internal node with the left node referring to the trie containing a key whose initial bit is 0 and right node 1 (with the leading bit considered to be removed for the purpose of constructing subtrees)
- a key exists in the leaf node defined by the digital path from the root to that node
- a search miss ends on a null leaf
- trie search:

```
private:
    Item searchR(link h, Key v, int d) {
        if (h == 0) return nullItem;
        if (h->l == 0 && h->r == 0) {
            Key w = h->item.key();
            return (v == w) ? h->item : nullItem;
```

```
        }
        if (digit(v, d) == 0)
           return searchR(h->l, v, d+1);
        else return searchR(h->r, v, d+1);
    }
public:
    Item search(Key v) { return searchR(head, v, 0); }
```

- trie insertion:

```
private:
    link split(link p, link q, int d) {
        link t = new node(nullItem); t->N = 2;
        Key v = p->item.key(); Key w = q->item.key();
        switch(digit(v, d)*2 + digit(w, d)) {
            case 0: t->l = split(p, q, d+1); break;
            case 1: t->l = p; t->r = q; break;
            case 2: t->r = p; t->l = q; break;
            case 3: t->r = split(p, q, d+1); break;
        }
        return t;
    }
    void insertR(link& h, Item x, int d) {
        if (h == 0) { h = new node(x); return; }
        if (h->l == 0 && h->r == 0) {
            h = split(new node(x), h, d);
            return;
        }
        if (digit(x.key(), d) == 0)
            insertR(h->l, x, d+1);
        else insertR(h->r, x, d+1);
    }
public:
    ST(int maxN) { head = 0; }
    void insert(Item item) { insertR(head, item, 0); }
```

- properties:
  - the structure of a trie is independent of the key insertion order -> there is a unique trie for any given set of distinct keys
  - insertion or search for a random key in a trie built from N random (distinct) bitstrings requires about lg(N) bit comparisons on the average. THe worst-case number of bit comparisons is bounded only by the number of bits in the search key.
  - a trie built from N random w-bit keys has about N/ln(2) or about 1.44 * N nodes on the average

Patricia Tries

- basic trie search has two main flaws:
  - has many extra nodes to support the path
  - has two distinct types of nodes

- patricia (practical algorithm to retrieve information coded in alphanumeric) tries::
  - trie with N keys and only has N nodes and requires lg(N) comparisons and one full key comparison per search
- each node stores an index in the bit string that dictates which path to take out of the node
- can regard standard tries and patricia tries as different representations of the same structure
- patricia trie search:

```
private:
    Item searchR(link h, Key v, int d) {
        if (h->bit <= d) return h->item;
        if (digit(v, h->bit) == 0)
           return searchR(h->l, v, h->bit);
        else return searchR(h->r, v, h->bit);
    }
public:
    Item search(Key v) {
        Item t = searchR(head, v, -1);
        return (v == t.key()) ? t : nullItem;
    }
```

- patricia trie insertion:

```
private:
    link insertR(link h, Item x, int d, link p) {
        Key v = x.key();
        if ((h->bit >= d) || (h->bit <= p->bit)) {
            link t = new node(x); t->bit = d;
            t->l = (digit(v, t->bit) ? h : t);
            t->r = (digit(v, t->bit) ? t : h);
            return t;
        }
        if (digit(v, h->bit) == 0)
           h->l = insertR(h->l, x, d, h);
        else h->r = insertR(h->r, x, d, h);
        return h;
    }
public:
    void insert(Item x) {
        Key v = x.key(); int i;
        Key w = searchR(head->l, v, -1).key();
        if (v == w) return;
        for (i = 0; digit(v, i) == digit(w, i); i++) ;
        head->l = insertR(head->l, x, i, head);
    }
    Symbol_table(int maxN) {
        head = new node(nullItem);
        head->l = head->r = head;
    }
```

- patricia trie search:

```
private:
    void showR(link h, ostream& os, int d) {
        if (h->bit <= d) { h->item.show(os); return; }
        showR(h->l, os, h->bit);
        showR(h->r, os, h->bit);
    }
public:
    void show(ostream& os) { showR(head->l, os, -1); }
```

- properties:
  - insertion or search for a random key in a patricia trie built from N random bitstrings requires about lg(N) bit comparisons on the average and about 2 * lg(N) bit comparisons in the worst case.
  - the number of bit comparisons is never worse than the length of the key
- tries, digital search trees, and patricia tries are competitive with balanced trees
- prefer patricia tries when search keys are long

Multiway Tries and TSTs - TODO: spend more time here

- radix sorting performance was improved by considering more than 1 bit at a time, this also works for radix search
- there is an important additional consideration -> considering r bits at a time corresponds to using tree nodes with R = 2 ^r links which results in a lot of wasted space
- definition:
  - existence trie: trie for empty set of keys is a null link, and the non-empty set of keys represented by an internal node with links referring to the trie for each possible key digit, with the leading digit removed for constructing subtrees
  - multiway trie: trie for a single key is a leaf containing the key, trie for a set of keys of cardinality greater than one is an internal node with links referring to tries for keys with each possible digit value, with the leading digit removed for constructing subtrees
- existence trie search and insertion:

```
private:
    struct node {
        node **next;
        node() {
            next = new node*[R];
            for (int i = 0; i < R; i++) next[i] = 0;
        }
    };
    typedef node *link;
    link head;
    Item searchR(link h, Key v, int d) {
        int i = digit(v, d);
        if (h == 0) return nullItem;
        if (i == NULLdigit) {
            Item dummy(v); return dummy;
        }
```

```
        return searchR(h->next[i], v, d+1);
    }
    void insertR(link& h, Item x, int d) {
        int i = digit(x.key(), d);
        if (h == 0) h = new node;
        if (i == NULLdigit) return;
        insertR(h->next[i], x, d+1);
    }
public:
    ST(int maxN) { head = 0; }
    Item search(Key v) { return searchR(head, v, 0); }
    void insert(Item x) { insertR(head, x, 0); }
```

- existence TST search and insertion:

```
private:
    struct node {
        Item item; int d; node *l, *m, *r;
        node(int k) { d = k; l = 0; m = 0; r = 0; }
    };
    typedef node *link;
    link head;
    Item nullItem;
    Item searchR(link h, Key v, int d) {
        int i = digit(v, d);
        if (h == 0) return nullItem;
        if (i == NULLdigit) { Item dummy(v); return dummy; }
        if (i < h->d) return searchR(h->l, v, d);
        if (i == h->d) return searchR(h->m, v, d+1);
        if (i > h->d) return searchR(h->r, v, d);
    }
    void insertR(link& h, Item x, int d) {
        int i = digit(x.key(), d);
        if (h == 0) h = new node(i);
        if (i == NULLdigit) return;
        if (i < h->d) insertR(h->l, x, d);
        if (i == h->d) insertR(h->m, x, d+1);
        if (i > h->d) insertR(h->r, x, d);
    }
public:
    ST(int maxN) { head = 0; }
    Item search(Key v) { return searchR(head, v, 0); }
    void insert(Item x) { insertR(head, x, 0); }
```

- properties:
  - search or insertion in a standard r-ary trie requires about log sub R (N) comparisons on average in a tree built from N random bytestrings. The number of links in an r-ary trie built from N random keys is about (R * N) / ln(R). The number of byte comparisons for search or insertion is no more than the number of bytes in the search key
  - a search or insertion in a full TST requires time proportional to the key length. The number of links in a TST is at most three times the number of characters in all the keys.

    - a search or insertion in a TST with items in leaves (no one-way branching at the bottom) and R^t-way branching at the root requires roughly ln(N - t) * ln(R) byte accesses for N keys that are random bytestrings. The number of links required is R ^ t for the root node plus a small constant time N.
- TSTs are good because they deal with key irregularities very well
    - most practical applications deal with large character sets with non-uniform usage
    - search misses are extremely efficient, even for long keys
- partial match searching in TSTs

```
private:
    char word[maxW];
    void matchR(link h, char *v, int i) {
        if (h == 0) return;
        if ((*v == 0) && (h->d == 0)) { word[i] = 0; cout << word << " "; }
        if ((*v == '*') || (*v == h->d)) { word[i] = h->d; matchR(h->m, v+1, i+1);
        if ((*v == '*') || (*v < h->d)) matchR(h->l, v, i);
        if ((*v == '*') || (*v > h->d)) matchR(h->r, v, i);
      }
public:
    void match(char *v) { matchR(head, v, 0); }
```

- hybrid TST node type definitions:

```
struct node {
    Item item; int d; node *l, *m, *r;
    node(Item x, int k) { item = x; d = k; l = 0; m = 0; r = 0; }
    node(node* h, int k) { d = k; l = 0; m = h; r = 0; }
    int internal() { return d != NULLdigit; }
};
typedef node *link;
link heads[R];
Item nullItem;
```

- hybrid TST insertion for symbol table type

```
private:
    link split(link p, link q, int d) {
        int pd = digit(p->item.key(), d), qd = digit(q->item.key(), d);
        link t = new node(nullItem, qd);
        if (pd < qd) { t->m = q; t->l = new node(p, pd); }
        if (pd == qd) { t->m = split(p, q, d+1); }
        if (pd > qd) { t->m = q; t->r = new node(p, pd); }
        return t;
      }
    link newext(Item x) { return new node(x, NULLdigit); }
    void insertR(link& h, Item x, int d) {
        int i = digit(x.key(), d);
        if (h == 0) { h = new node(newext(x), i); return; }
        if (!h->internal()) { h = split(newext(x), h, d); return; }
        if (i < h->d) insertR(h->l, x, d);
```

```
        if (i == h->d) insertR(h->m, x, d+1);
        if (i > h->d) insertR(h->r, x, d);
      }
public:
    ST(int maxN) { for (int i = 0; i < R; i++) heads[i] = 0; }
    void insert(Item x) { insertR(heads[digit(x.key(), 0)], x, 1); }
```

- hybrid TST search for symbol-table type

```
private:
    Item searchR(link h, Key v, int d) {
        if (h == 0) return nullItem;
        if (h->internal()) {
            int i = digit(v, d), k = h->d;
            if (i < k) return searchR(h->l, v, d);
            if (i == k) return searchR(h->m, v, d+1);
            if (i > k) return searchR(h->r, v, d);
        }
        if (v == h->item.key()) return h->item;
        return nullItem;
    }
public:
    Item search(Key v) { return searchR(heads[digit(v, 0)], v, 1); }
```

Text-String-Index Algorithms

- suffix tree: a search tree built from keys defined by string pointer into a text string
- tries work well as opposed to hashing because hashing algorithms have a running time dependent on the key length
- patricia tries work well
- TSTs provide many of the same advantages of patricia tries with some additional
- for most text indexing applications, the text is fixed so dynamic insert does not need to be supported and binary search might be suitable
- just relying on binary search for fixed text indexing reduces the number of pointers required to index into the text string

External Searching

- need algorithms that are appropriate for searching in large external text files
- some considerations:
  - tape-like devices are limited to sequential access
  - disk-like random access facilitates search and insert
  - may have access to a large virtual memory

    - direct relationship to database functionality

Rules of the Game

- model:
    - consider the access to be through pages or contiguous blocks of information that can be accessed efficiently by the hardware
    - directly relevant to a filesystem, virtual memory, and caching
- definitions:
    - page: a contiguous block of data
    - probe: first access to a page
- many additional considerations for long-lived databases not considered here

Indexed Sequential Access

- basic implementation -> keep an array with keys and item references and use binary search
- due to page access:
    - binary tree with keys and page pointers in internal nodes and keys and item pointers in external nodes
    - can use an m-ary tree to reduce access costs
- this model is similar to device dependent hardware access implementations
- properties:
    - a search in an indexed sequential file requires only a constant number of probes, but an insertion can require that the entire index be rebuilt
- modifying the directory (indexed) is expensive

B Trees - TODO: more notes here

- multiway trees with a relaxed requirement of the number of entries in each node supports the indexed sequential access requirements with the added benefit of dynamic growth
- generalization of the 2-3-4 structure (i.e. 4-5-6-7-8 etc)
- definition:
    - b tree (of order M): a tree that either is empty or is made up of "k-nodes", with k - 1 keys and k links to trees representing each of the k intervals delimited by the keys. k must be between 2 and M at the root and M / 2 and M at every other node. All links to empty trees must be the same distance from the root.
- b tree node type definition:

```
template <class Item, class Key>
struct entry { Key key; Item item; struct node *next; };
struct node {
    int m; entry<Item, Key> b[M];
    node() { m = 0; }
```

```
};
typedef node *link;
```

- b tree search:

```
private:
  Item searchR(link h, Key v, int ht)
    { int j;
      if (ht == 0)
        for (j = 0; j < h->m; j++)
          { if (v == h->b[j].key)
              return h->b[j].item; }
      else
        for (j = 0; j < h->m; j++)
          if ((j+1 == h->m) || (v < h->b[j+1].key))
            return searchR(h->b[j].next, v, ht-1);
      return nullItem;
    }
public:
  Item search(Key v)
    { return searchR(head, v, HT); }
```

- b tree insertion:

```
private:
  link insertR(link h, Item x, int ht)
    { int i, j; Key v = x.key(); entry<Item, Key> t;
      t.key = v; t.item = x;
      if (ht == 0)
        for (j = 0; j < h->m; j++)
          { if (v < h->b[j].key) break; }
      else
        for (j = 0; j < h->m; j++)
          if ((j+1 == h->m) || (v < h->b[j+1].key))
            { link u;
              u = insertR(h->b[j++].next, x, ht-1);
              if (u == 0) return 0;
              t.key = u->b[0].key; t.next = u;
              break;
            }
     for (i = h->m; i > j; i--) h->b[i] = h->b[i-1];
     h->b[j] = t;
     if (++h->m < M) return 0; else return split(h);
    }
public:
  ST(int maxN)
    { N = 0; HT = 0; head = new node; }
  void insert(Item item)
    { link u = insertR(head, item, HT);
      if (u == 0) return;
      link t = new node(); t->m = 2;
      t->b[0].key = head->b[0].key;
      t->b[1].key =  u->b[0].key;
```

```
    t->b[0].next = head; t->b[1].next = u;
    head = t; HT++;
  }
```

- b tree node split:

```
{ link t = new node();
  for (int j = 0; j < M/2; j++)
    t->b[j] = h->b[M/2+j];
  h->m = M/2; t->m = M/2;
  return t;
}
```

- properties:
  - a search or an insertion in a b tree of order M with N items requires between log sub M (N) and log sub (M/2) (N) probes (constant for practical purposes)
  - a b tree of order M constructed from N random items is expected to have about 1.44 * (N/M) pages

Perspective

- these algorithms cannot be used directly for implementing a database or a filesystem
  - the parameters need to be tuned for the system
  - need reliability and error detection and correction
- virtual memory can facilitate the algorithm use for in memory operations on a large dataset but size considerations will dictate what is brought into cache and how data is split across pages