COMP1927 Final Exam

Algorithms Almanac

Topic Index: Linear ADTs ... Sorting ... Searching ... Graphs

This document gives code for representative ADTs and algorithms discussed in Lectures. It's organised as a single long page, broken down by topic area. The code assumes that all collection types are made up of `Items`, where the `Item` type has equality, less than, greater than, comparison, and display functions available as `eq(it1,it2)`, `less(it1,it2)`, `greater(it1,it2)`, `cmp(it1,it2)` and `show(it)`. Sometimes, `Items` have keys which are integer or string values that uniquely identify the `Item`. They key value can be extracted from an `Item` using the `key(it)` function. ADTs are implemented via a pointer to a hidden representation type (e.g. the `Stack` type is a pointer to a `StackRep` structure).

Note: the code below has not been compiled; we make no guarantees that there are no bugs. This code is provided simply as a guide to how the algorithms were implemented.

Linear ADTs (Stacks ... Queues ... Lists)

Stack ADT

Interface:

```
typedef struct StackRep *Stack;
// set up empty stack
Stack newStack(int);
// remove unwanted stack
void dropStack(Stack);
// insert an Item on top of stack
void StackPush(Stack,Item);
// remove Item from top of stack
Item StackPop(Stack);
// check whether stack is empty
Bool StackIsEmpty(Stack);
```

Implementation (via arrays):

```
typedef struct StackRep {
   Item *item;
   int top;
} StackRep;
// set up empty stack
Stack newStack(int n)
{
   Stack s;
   s = malloc(sizeof(StackRep));
   assert(s != NULL);
   s->item = malloc(n * sizeof(Item));
   assert(s->item != NULL);
   s->top = -1;
   return s;
}
// remove unwanted stack
void dropStack(Stack s)
{
   assert(s != NULL);
   free(s->item);
   free(s);
}
// insert Item on top of stack
void StackPush(Stack s, Item it)
```

```
{
   assert(s->top < MAXITEMS-1);
   s->top++;
   int i = s->top;
   s->item[i] = it;
}
// remove Item from top of stack
Item StackPop(Stack s)
{
   assert(s->top > -1);
   int i = s->top;
   Item it = s->item[i];
   s->top--;
   return it;
}
// check whether stack is empty
Bool StackIsEmpty(Stack s)
{
   return (s->top < 0);
}
```

Queue ADT

Interface:

```
typedef struct QueueRep *Queue;
// set up empty queue
Queue newQueue(int);
// remove unwanted queue
void dropQueue(Queue);
// insert Item at back of queue
void QueueJoin(Queue,Item);
// remove Item from front of queue
Item QueueLeave(Queue);
// check whether queue is empty
Bool QueueIsEmpty(Queue);
```

Implementation (via arrays):

```
typedef struct QueueRep {
   Item *item;  // array to hold Items
   int front;  // next Item to be removed
   int back;  // last Item added
   int nitems;  // # Items currently in queue
   int maxitems;  // size of array
} QueueRep;
// set up empty queue
Queue newQueue(int n)
{
   Queue q;
   q = malloc(sizeof(QueueRep));
   assert(q != NULL);
   q->item = malloc(n * sizeof(Item));
   assert(q->item != NULL);
   q->front = q->back = 0;
   q->nitems = 0; q->maxitems = n;
   return q;
}
// remove unwanted queue
void dropQueue(Queue q)
{
   assert(q != NULL);
```

```
   free(q->item);
   free(q);
}
// insert item on top of queue
void QueueJoin(Queue q, Item it)
{
   assert(q->nitems < q->maxitems);
   q->item[q->front] = it;
   q->nitems++;
   q->front = (q->front+1)%q->maxitems;
}
// remove item from front of queue
Item QueueLeave(Queue q)
{
   assert(q->nitems > 0);
   Item it = q->item[q->back];
   q->nitems--;
   q->back = (q->back+1)%q->maxitems;
   return it;
}
// check whether queue is empty
Bool QueueIsEmpty(Queue q)
{
   return (q->nitems == 0);
}
```

List ADT

Interface:

```
typedef struct ListRep *List;
// create a new empty List
List newList();
// free up all space associated with list
void freeList(List);
// display list as one Item per line on stdout
void showList(List);
// apppend one Item to the end of a list
void ListInsert(List, int);
// delete first occurrence of v from a list
// if v does not occur in List, no effect
void ListDelete(List, Item);
// return number of elements in a list
int ListLength(List);
```

Implementation (via linked list):

```
typedef struct ListNode {
   int  data;  // value of this list item
   struct ListNode *next;
               // pointer to node containing next element
} ListNode;
typedef struct ListRep {
   int  size;  // number of elements in list
   ListNode *first;  // node containing first value
   ListNode *last;  // node containing last value
} ListRep;
// create a new empty List
List newList()
{
   ListRep *L;
   L = malloc(sizeof(ListRep));
```

```
   assert (L != NULL);
   L->size = 0;
   L->first = NULL;
   L->last = NULL;
   return L;
}
// free up all space associated with list
void freeList(List L)
{
   ListNode *curr, *next;
   assert(L != NULL);
   curr = L->first;
   while (curr != NULL) {
      next = curr->next;
      free(curr);
      curr = next;
   }
   free(L);
}
// display list as one integer per line on stdout
void showList(List L)
{
   ListNode *curr;
   curr = L->first;
   while (curr != NULL) {
      show(curr->data); printf("\n");
      curr = curr->next;
   }
}
// create a new ListNode with value it
// (this function is local to this ADT)
static ListNode *newListNode(Item it)
{
   ListNode *n;
   n = malloc(sizeof(ListNode));
   assert(n != NULL);
   n->data = item;
   n->next = NULL;
   return n;
}
// apppend one Item to the end of a list
void ListInsert(List L, Item it)
{
   assert(L != NULL);
   ListNode *n;
   n = newListNode(it);
   if (L->first == NULL)
      L->first = L->last = n;
   else {
      L->last->next = n;
      L->last = n;
   }
   L->size++;
}
// remove an item from a List
void ListDelete(List L, Item it)
{
   assert(L != NULL);
   ListNode *curr, *prev;
   prev = NULL; curr = L->first;
   while (curr != NULL && !eq(curr->data,it)) {
      prev = curr;
      curr = curr->next;
```

```
   }
   if (curr == NULL) return;
   if (prev == NULL)
      L->first = curr->next;
   else
      prev->next = curr->next;
   if (L->last == curr)
      L->last = prev;
   L->size--;
   free(curr);
}
// return number of elements in a list
int ListLength(List L)
{
   assert(L != NULL);
   return L->size;
}
```

Priority Queue:

Interface:

```
typedef struct PQueueRep *PQueue;
create new empty priority queue
PQueue newPQueue(int size);
add item to priority queue
void PQJoin(PQueue q, Item it);
remove item from priority queue
Item PQLeave(PQueue q);
free up priority queue
void dropPQueue(PQueue q);
```

Implementation:

```
struct PQueueRep {
    int   nItems; // count of items
    Item *items;  // heap-array of Items
    int   size;   // size of array
}
// create a new empty queue
PQueue newPQueue(int size)
{
    PQueue q = malloc(sizeof(struct PQrep));
    assert(q != NULL);
    // indexes start from 1
    q->items = malloc(sizeof(Item) * (size+1));
    assert(q->items != NULL);
    q->nItems = 0;
    q->size = size;
    return q;
}
// add a new item into the queue
void PQJoin(PQueue q, Item it)
{
    assert(q != NULL && q->nItems < q->size);
    q->nItems++;
    q->items[q->nItems] = it;
    fixUp(q->items, q->nItems);
}
// remove item from priority queue
Item PQLeave(PQueue q)
{
    assert(q != NULL && q->nItems > 0);
```

```
    swap(q->items, 1, q->nItems);
    q->nItems--;
    fixDown(p->items, 1, q->nItems);
    return q->items[q->nItems+1];
}
free up priority queue
void dropPQueue(PQueue q)
{
    assert(q != NULL);
    free(q->items);
    free(q);
}
void fixUp(Item a[], int k)
{
   while (k > 1 && less(a[k/2],a[k])) {
      swap(a, k, k/2);
      k = k/2;  // integer division
   }
}
void fixDown(Item a[], int k, int N)
{
    while (2*k <= N) {
    int j = 2*k;
    // choose larger of two children
    if (j < N && less(a[j], a[j+1])) j++;
        if (!less(a[k], a[j])) break;
        swap(a, k, j);
        k = j;
    }
}
```

Sorting (O(n^2) Algs ... O(n log n) Algs)

O(n^2) Sorting Algorithms

Sorting problem:

```
Pre-condition: a[0..N-1] contain Items
Post-condition: forall i:0..N-2,  key(a[i]) ≤ key(a[i+1])

Stability: consider item1 and item2 where key(item1) == key(item2)
if, before sorting, item1 is a[i] && item2 is a[j] && i < j
then, after sorting, item1 is a[m] && item2 is a[n] && m < n
```

Selection sort:

```
void selectionSort(int a[], int lo, int hi)
{
   int i, j, min;
   for (i = lo; i < hi; i++) {
      min = i;
      for (j = i+1; j <= hi; j++) {
         if (less(a[j],a[min])) min = j;
      }
      swap(a[i], a[min]);
   }
}
```

Bubble sort:

```
void bubbleSort(int a[], int lo, int hi)
{
   int i, j, nswaps;
   for (i = lo; i < hi; i++) {
      nswaps = 0;
      for (j = hi; j > i; j--) {
         if (less(a[j], a[j-1])) {
            swap(a[j], a[j-1]);
            nswaps++;
         }
      }
      if (nswaps == 0) break;
   }
}
```

Insertion sort:

```
void insertionSort(int a[], int lo, int hi)
{
   int i, j, val;
   for (i = lo+1; i <= hi; i++) {
      val = a[i];
      for (j = i; j > lo; j--) {
         if (!less(val,a[j-1])) break;
         a[j] = a[j-1];
      }
      a[j] = val;
   }
}
```

O(n log n) Sorting Algorithms

Quicksort: (with median-of-three partitioning)

```
void quicksort(Item a[], int lo, int hi)
{
   int i; // index of pivot
   if (hi <= lo) return;
   medianOfThree(a, lo, hi);
   i = partition(a, lo+1, hi-1);
   quicksort(a, lo, i-1);
   quicksort(a, i+1, hi);
}
void medianOfThree(Item a[], int lo, int hi)
{
   int mid = (lo+hi)/2;
   if (less(a[mid],a[lo])) swap(a, lo, mid);
   if (less(a[hi],a[mid])) swap(a, mid, hi);
   if (less(a[mid],a[lo])) swap(a, lo, mid);
   // now, we have a[lo] ≤ a[mid] ≤ a[hi]
   // swap a[mid] to a[lo+1] to use as pivot
   swap(a, lo+1, mid);
}
int partition(Item a[], int lo, int hi)
{
   Item v = a[lo];  // pivot
   int  i = lo+1, j = hi;
   for (;;) {
      while (less(a[i],v) && i < j) i++;
      while (less(v,a[j]) && j > i) j--;
      if (i == j) break;
      swap(a,i,j);
```

```
    }
    j = less(a[i],v) ? i : i-1;
    swap(a,lo,j);
    return j;
}
```

Mergesort:

```
void mergesort(Item a[], int lo, int hi)
{
    int mid = (lo+hi)/2; // mid point
    if (hi <= lo) return;
    mergesort(a, lo, mid);
    mergesort(a, mid+1, hi);
    merge(a, lo, mid, hi);
}
void merge(Item a[], int lo, int mid, int hi)
{
    int  i, j, k, nitems = hi-lo+1;
    Item *tmp = malloc(nitems*sizeof(Item));

    i = lo; j = mid+1; k = 0;
    // scan both segments, copying to tmp
    while (i <= mid && j <= hi) {
      if (less(a[i],a[j]))
         tmp[k++] = a[i++];
      else
         tmp[k++] = a[j++];
    }
    // copy items from unfinished segment
    while (i <= mid) tmp[k++] = a[i++];
    while (j <= hi) tmp[k++] = a[j++];

    //copy tmp back to main array
    for (i = lo, k = 0; i <= hi; i++, k++)
       a[i] = tmp[k];
    free(tmp);
}
```

Heapsort: (require priority queue PQueue type)

```
void HeapSort(Item a[], int lo, int hi)
{
    PQueue pq = newPQueue(hi-lo+1);
    int i;
    for (i = lo; i <= hi; i++) {
       PQJoin(pq, a[i]);
    }
    for (i = hi; i >= lo; i--) {
       Item it = PQLeave(pq);
       a[i] = it;
    }
}
```

Searching (Binary Search ... BSTs ... Balanced Trees ... Hashing)

Binary Search

Binary search in array:

```
// search for key k in array a[]
// - returns index of location for k
```

```
// - doesn't indicate whether key is actually there or not
int findInArray(Key k, Item a[], int lo, int hi)
{
    if (hi <= lo) return lo;
    int mid = (hi+lo)/2;
    int diff = cmp(k, key(a[mid]));
    if (diff < 0)
        return findInArray(k, a, lo, mid);
    else if (diff > 0)
        return findInArray(k, a, mid+1, hi);
    else
        return mid;
}
```

Binary Search Trees (BSTs)

BST Interface:

```
typedef struct Node *Tree;
// create an empty Tree
Tree newTree();
// free memory associated with Tree
void dropTree(Tree);
// display a Tree (sideways)
void showTree(Tree);
// insert a new item into a Tree
Tree TreeInsert(Tree, Item);
Tree TreeInsertAtRoot(Tree, Item);
Tree TreeInsertRandom(Tree, Item);
// delete item with given key from Tree
Tree TreeDelete(Tree, Key);
// check whether item with given key is in Tree
int TreeFind(Tree, Key);
// compute depth of Tree
int TreeDepth(Tree);
// count #nodes in Tree
int TreeNumNodes(Tree);
// fetch i'th (from left) item from Tree
Item *get_ith(Tree, int);
// partition Tree around i'th Item
Tree partition(Tree, int);
// rotate Tree left/right around root
Tree rotateR(Tree);
Tree rotateL(Tree);
```

BST Implementation:

```
typedef struct Node *Link;
typedef struct Node {
    Item value;
    Link left, right;
} Node;
// make a new node containing an Item
static Link newNode(Item it)
{
    Link new = malloc(sizeof(Node));
    assert(new != NULL);
    new->value = it;
    new->left = new->right = NULL;
    return new;
}
// create a new empty Tree
```

```
Tree newTree()
{
    return NULL;
}
// free memory associated with Tree
void dropTree(Tree t)
{
    if (t == NULL) return;
    dropTree(t->left);
    dropTree(t->right);
    free(t);
}
// display a Tree (sideways)
void showTree(Tree t)
{
    void doShowTree(Tree);
    doShowTree(t);
}
// insert a new value into a Tree
Tree TreeInsert(Tree t, Item it)
{
    if (t == NULL) return newNode(it);
    int diff = cmp(key(it),key(t->value));
    if (diff == 0)
        t->value = it;
    else if (diff < 0)
        t->left = TreeInsert(t->left, it);
    else if (diff > 0)
        t->right = TreeInsert(t->right, it);
    return t;
}
// insert a new value as root of Tree
Tree insertAtRoot(Tree t, Item it)
{
   if (t == NULL) return newNode(it);
   int diff = cmp(key(it), key(t->value));
   if (diff == 0)
      t->value = it;
   else if (diff < 0) {
      t->left = insertAtRoot(t->left, it);
      printf("rotateR(%d)\n",t->value);
      t = rotateR(t);
   }
   else if (diff > 0) {
      t->right = insertAtRoot(t->right, it);
      printf("rotateL(%d)\n",t->value);
      t = rotateL(t);
   }
   return t;
}
Tree insertRandom(Tree t, Item it)
{
   int size(Tree);

   if (t == NULL) return newNode(it);
   // 1 in 3 chance of doing root insert
   int chance = rand() % 3;
   if (chance == 0)
      t = insertAtRoot(t, it);
   else
      t = TreeInsert(t, it);
   return t;
}
```

```
// delete item with given key from Tree
Tree TreeDelete(Tree t, Key k)
{
    Tree deleteRoot(Tree);

    if (t == NULL)
        return NULL;
    int diff = cmp(k,key(t->value));
    if (diff == 0)
        t = deleteRoot(t);
    else if (diff < 0)
        t->left = TreeDelete(t->left, k);
    else if (diff > 0)
        t->right = TreeDelete(t->right, k);
    return t;
}
// delete root of tree
Tree deleteRoot(Tree t)
{
    Link newRoot;
    // if no subtrees, tree empty after delete
    if (t->left == NULL && t->right == NULL) {
        free(t);
        return NULL;
    }
    // if only right subtree, make it the new root
    else if (t->left == NULL && t->right != NULL) {
        newRoot = t->right;
        free(t);
        return newRoot;
    }
    // if only left subtree, make it the new root
    else if (t->left != NULL && t->right == NULL) {
        newRoot = t->left;
        free(t);
        return newRoot;
    }
    // else (t->left != NULL && t->right != NULL)
    // so has two subtrees
    // - find inorder successor (grab value)
    // - delete inorder successor node
    // - move its value to root
    Link parent = t;
    Link succ = t->right; // not null!
    while (succ->left != NULL) {
        parent = succ;
        succ = succ->left;
    }
    int succVal = succ->value;
    t = TreeDelete(t,succVal);
    t->value = succVal;
    return t;
}
// check whether item with given key is in Tree
int TreeFind(Tree t, Key k)
{
    if (t == NULL) return 0;
    int res, diff = cmp(k,key(t->value));
    if (diff < 0)
        res = TreeFind(t->left, k);
    else if (diff > 0)
        res = TreeFind(t->right, k);
    else // (diff == 0)
        res = 1;
```

```
    return res;
}
// compute depth of Tree
int TreeDepth(Tree t)
{
    if (t == NULL)
        return 0;
    else {
        int ldepth = TreeDepth(t->left);
        int rdepth = TreeDepth(t->right);
        //return 1 + (ldepth > rdepth)?ldepth:rdepth;
        if (ldepth > rdepth)
            return 1+ldepth;
        else
            return 1+rdepth;
    }
}
// count #nodes in Tree
int TreeNumNodes(Tree t)
{
    if (t == NULL) return 0;
    return 1 + TreeNumNodes(t->left)
             + TreeNumNodes(t->right);
}
// fetch i'th (from left) item from Tree
Item *get_ith(Tree t, int i)
{
   if (t == NULL) return NULL;
   assert(0 <= i && i < size(t));
   int n = size(t->left); // #nodes to left of root
   if (i < n) return get_ith(t->left, i);
   if (i > n) return get_ith(t->right, i-n-1);
   return &(t->value);
}
// // partition Tree around i'th Item
Tree partition(Tree t, int i)
{
   if (t == NULL) return NULL;
   assert(0 <= i && i < size(t));
   int n = size(t->left);
   if (i < n) {
      t->left = partition(t->left, i);
      t = rotateR(t);
   }
   if (i > n) {
      t->right = partition(t->right, i-n-1);
      t = rotateL(t);
   }
   return t;
}
// rotate Tree right around node
Link rotateR(Link n1)
{
   if (n1 == NULL) return n1;
   Link n2 = n1->left;
   if (n2 == NULL) return n1;
   n1->left = n2->right;
   n2->right = n1;
   return n2;
}
// rotate Tree left around node
Link rotateL(Link n2)
{
```

```
   if (n2 == NULL) return n2;
   Link n1 = n2->right;
   if (n1 == NULL) return n2;
   n2->right = n1->left;
   n1->left = n2;
   return n1;
}
```

Note that the above has operations relevant for balancing.
The tree types below are specificaly designed to be balance.

Balanced Trees

Splay Trees:

```
#define L left
#define R right
// Other operations are as for BSTs
Tree insertSplay(Tree t, Item it)
{
   if (t == NULL) return newNode(it);
   int diff = cmp(key(it), key(t->value));
   if (diff == 0)
      t->value = it;
   else if (diff < 0) {
      if (t->L == NULL) {
         t->L = newNode(it);
         t->nnodes++;
      }
      int ldiff = cmp(key(it), key(t->L->value));
      if (ldiff < 0) {
         // Case 1: left-child of left-child
         t->L->L = insertSplay(t->L->L, it);
         t->L->nnodes++;
         t->nnodes++;
         t = rotateR(t);
      }
      else if (ldiff > 0) {
         // Case 2: right-child of left-child
         t->L->R = insertSplay(t->L->R, it);
         t->L->nnodes++;
         t->nnodes++;
         t->L = rotateL(t->L);
      }
      return rotateR(t);
   }
   else if (diff > 0) {
      if (t->R == NULL) {
         t->R = newNode(it);
         t->nnodes++;
      }
      int rdiff = cmp(key(it), key(t->R->value));
      if (rdiff < 0) {
         // Case 3: left-child of right-child
         t->R->L = insertSplay(t->R->L, it);
         t->R->nnodes++;
         t->nnodes++;
         t->R = rotateR(t->R);
      }
      else if (rdiff > 0) {
         // Case 4: right-child of right-child
         t->R->R = insertSplay(t->R->R, it);
         t->R->nnodes++;
```

```
         t->nnodes++;
         t = rotateL(t);
      }
      return rotateL(t);
   }
   else
      t->value = it;
   return t;
}
// search Tree for item with key k
Item *searchSplay(Tree *t, Key k)
{
   Link root = *t;
   if (root == NULL) return NULL;
   root = splay(root,k);
   *t = root;
   if (key(root->value) == k)
      return &(root->value);
   else
      return NULL;
}
```

AVL Trees:

```
// Other operations are as for BSTs
Tree insertAVL(Tree t, Item it)
{
   if (t == NULL) return newNode(it);
   int diff = cmp(key(it), key(t->value));
   if (diff == 0)
      t->value = it;
   else if (diff < 0) {
      t->left = insertAVL(t->left, it);
      t->nnodes = count(t);
   }
   else if (diff > 0) {
      t->right = insertAVL(t->right, it);
      t->nnodes = count(t);
   }
   int dL = depth(t->left);
   int dR = depth(t->right);
   if ((dL - dR) > 1) t = rotateR(t);
   else if ((dR - dL) > 1) t = rotateL(t);
   return t;
}
```

2-3-4 Tree Interface:

```
typedef struct node *Tree;
// Operations as for BST, except insert
```

2-3-4 Tree Implementation:

```
typedef struct node {
   int  order;   // 2, 3 or 4
   Item data[3];  // items in node
   Tree child[4]; // links to subtrees
} Node;

// create new 2-3-4 node
static Node *newNode(Item it)
{
```

```
   Node *new = malloc(sizeof(Node));
   assert(new != NULL);
   new->order = 2;
   new->data[0] = it;
   new->child[0] = new->child[1] = NULL;
   return new;
}
// search for item with key k
Item *search(Tree t, Key k)
{
   if (t == NULL) return NULL;
   int i;  int diff;  int nitems = t->order-1;
   // find relevant slot in items
   for (i = 0; i < nitems; i++) {
      diff = cmp(k, key(t->data[i]));
      if (diff <= 0) break;
   }
   if (diff == 0)
      // match; return result
      return &(t->data[i]);
   else
      // keep looking in relevant subtree
      return search(t->child[i], k);
}
//insert new Item into Tree
Tree insert(Tree t, Item it)
{
   // algorithm only ...
   find leaf node where Item belongs (via search)
   if not full (i.e. order < 4) {
      insert Item in this node, order++
   }
   else if node is full (i.e. contains 3 Items) {
      split into two 2-nodes as leaves
      promote middle element to parent
      insert item into appropriate leaf 2-node
      if parent is a 4-node {
         continue split/promote upwards
         if promote to root, and root is a 4-node {
            split root node
            add new root node
            promote middle item to new root
         }
      }
   }
}
```

Red-black Trees:

```
typedef enum {RED,BLACK} Colr;
typedef struct Node *Link;
typedef struct Node *Tree;
typedef struct Node {
   Item data;   // actual data
   Colr colour; // colour of link to parent
   Tree left;   // left subtree
   Tree right;  // right subtree
} Node;
// make new node to hold supplied Item
Node *newNode(Item it, Colr c)
{
   Node *new = malloc(sizeof(Node));
   assert(new != NULL);
```

```
   new->data = it; new->colour = c;
   new->left = new->right = NULL;
   return new;
}
// search for Item with given key
Item *search(Tree t, Key k)
{
   if (t == NULL) return NULL;
   int diff = cmp(k, key(t->data));
   if (diff < 0)
      return search(t->left, k);
   else if (diffs > 0)
      return search(t->right, k);
   else // matches
      return &(t->data);
}
// insert new Item into tree
#define L left
#define R right
#define isRed(t) ((t) != NULL && (t)->colour == RED)
void insert(Tree t, Item it)
{
   t->root = insertRB(t->root, it, 0);
   t->root->colour = RED;
}
Link insertRB(Link t, Item it, int inRight)
{
   if (t == NULL) return newNode(it,RED);
   // node is a 4-node; lift it
   if (isRed(t->L) && isRed(t->R)) {
      t->colour = RED;
      t->L->colour = BLACK;
      t->R->colour = BLACK;
   }
   int diff = cmp(key(it),key(t->value));
   if (diff == 0)
      t->value = it;
   else if (diff < 0) {
      t->L = insertRB(t->L, it, 0);
      if (isRed(t) && isRed(t->L) && inRight)
         t = rotateR(t);
      if (isRed(t->L) && isRed(t->L->L)) {
         t = rotateR(t);
         t->colour = BLACK;
         t->R->colour = RED;
      }
   }
   else if (diff > 0) {
      t->R = insertRB(t->R, it, 1);
      if (isRed(t) && isRed(t->R) && !inRight)
         t = rotateL(t);
      if (isRed(t->R) and isRed(t->R->R) {
         t = rotateL(t);
         t->colour = BLACK;
         t->L->colour = RED;
      }
   }
   return t;
}
// other operations as for BSTs
```

Hashing

Hash table Interface:

```
typedef struct HashTabRep *HashTable;
// create an empty HashTable
HashTable newHashTable(int);
// free memory associated with HashTable
void dropHashTable(HashTable);
// insert a new value into a HashTable
void hashTableInsert(HashTable, Item);
// delete a value from a HashTable
void hashTableDelete(HashTable, Key);
// get Item from HashTable using Key
Item *hashTableSearch(HashTable, Key);
```

Hash Table Implementation: (separate chains)

```
#include "List.h"  // use Lists of Items
typedef struct HashTabRep {
   List *lists;  // lists of Items
   int   nslots; // # elements in array
   int   nitems; // # items stored in HashTable
} HashTabRep;
// convert key to index
static int hash(Key k, int N)
{
   int h = ... convert key to int
   return h % N;
}
// create an empty HashTable
HashTable newHashTable(int N)
{
   HashTable new = malloc(sizeof(HashTable));
   assert(new != NULL);
   new->lists = malloc(N*sizeof(List));
   assert(new->items != NULL);
   int i;
   for (i = 0; i < N; i++)
      new->lists[i] = newList();
   new->nslots = N;
   new->nitems = 0;
   return new;
}
// free memory associated with HashTable
void dropHashTable(HashTable ht)
{
   free(ht->lists);
   free(ht);
}
// insert a new value into a HashTable
void hashTableInsert(HashTable ht, Item it)
{
   Key k = key(it);
   int i = hash(k, ht->nslots);
   ListInsert(ht->lists[i], it);
}
// delete a value from a HashTable
void hashTableDelete(HashTable ht, Key k)
{
   int i = hash(k, ht->nslots);
   ListDelete(ht->lists[i], k);
}
// get Item from HashTable using Key
Item *hashTableSearch(HashTable ht, Key k)
```

```
{
   int i = hash(k, ht->nslots);
   return ListSearch(ht->lists[i], k);
}
```

Hash Table Implementation: (linear probing)

```
#include "List.h"  // use Lists of Items
typedef struct HashTabRep {
   Item *items;  // lists of Items
   int   nslots; // # elements in array
   int   nitems; // # items stored in HashTable
} HashTabRep;
// convert key to index
static int hash(Key k, int N)
{
   int h = ... convert key to int
   return h % N;
}
// create an empty HashTable
HashTable newHashTable(int N)
{
   HashTabRep *new = malloc(sizeof(HashTabRep));
   assert(new != NULL);
   new->items = malloc(N*sizeof(Item));
   assert(new->items != NULL);
   int i;
   for (i = 0; i < N; i++)
      new->items[i] = NoItem;
   new->nslots = N; new->nitems = 0;
   return new;

}
// free memory associated with HashTable
void dropHashTable(HashTable ht)
{
   free(ht->items);
   free(ht);
}
// insert a new value into a HashTable
void hashTableInsert(HashTable ht, Item it)
{
   int N = ht->nslots;
   Item *data = ht->items;
   Key k = key(it);
   int ix, j, i = hash(k,N);
   for (j = 0; j < N; j++) {
      ix = (i+j)%N;
      if (cmp(k,key(data[ix]) == 0)
         break;
      else if (data[ix] == NoItem)
         break;
   }
   if (j < N) {
      data[ix] = it;
      ht->nitems++;
   }
}
// delete a value from a HashTable
void hashTableDelete(HashTable ht, Key k)
{
   int N = ht->nslots;
   Item *data = ht->items;
```

```
   int j, i = hash(k,N);
   for (j = 0; j < N; j++) {
      int ix = (i+j)%N;
      if (cmp(k,key(data[ix]) == 0)
         break;
      else if (data[ix] == NoItem)
         return; // k not in table
   }
   data[ix] = NoItem;
   ht->nitems--;
   // clean up probe path
   j = ix+1;
   while (data[j] != NoItem) {
      Item it = data[j];
      data[j] = NoItem;
      ht->nitems--;
      insert(ht, it);
      j = (j+1)%N);
   }
}
// get Item from HashTable using Key
Item *hashTableSearch(HashTable ht, Key k)
{
   int N = ht->nslots;
   Item *data = ht->items;
   int j, i = hash(k,N);
   for (j = 0; j < N; j++) {
      int ix = (i+j)%N;
      if (cmp(k,key(data[ix]) == 0)
         return &(data[ix]);
   }
   return NULL;

}
```

Graphs

(Representation ... Traversal ... Digraphs ... Weighted Graphs)

Representation

Graph Interface:

```
// visible data structures for Graphs
typedef struct GraphRep *Graph;
// vertices denoted by integers 0..N-1
typedef int Vertex;
// edges are pairs of vertices (end-points)
typedef struct { Vertex v; Vertex w; } Edge;
// auxiliary operations on graphs
int  validV(Graph,Vertex); // validity check
Edge mkEdge(Graph, Vertex, Vertex); // edge creation
int  neighbours(Graph, Vertex, Vertex); // edge existence
// core operations on graphs
// make new graph with nV vertices
Graph newGraph(int nV);
// free memory allocated to graph
void  dropGraph(Graph);
// show "printable" representation of graph
void  showGraph(Graph);
// add new edge to a graph
void  insertE(Graph, Edge);
```

```
// remove an edge from a graph
void  removeE(Graph, Edge);
// returns #vertices & array of edges
int   edges(Graph, Edge *, int);
```

Implementation of Auxiliary Operations:

```
// is a vertex valid in a given Graph?
static int validV(Graph g, Vertex v)
{
   return (g != NULL && v >= 0 && v < g->nV);
}
// make an Edge value
Edge mkEdge(Graph g, Vertex v, Vertex w)
{
   assert(validV(g,v) && validV(g,w));
   Edge e = {v,w}; // struct assignment
   return e;
}
```

Adjacency Matrix Representation:

```
typedef struct GraphRep {
   int    nV;    // #vertices
   int    nE;    // #edges
   Bool **edges; // matrix of booleans
} GraphRep;
// check whether two vertices are connected
int neighbours(Graph g, Vertex v, Vertex w)
{
        assert(validV(g,v) && validV(g,w));
        return g->edges[v][w];
}
// make new graph with nV vertices
Graph newGraph(int nV)
{
   assert(nV >= 0);
   int i, j;
   int **e = malloc(nV * sizeof(int *));
   assert(e != NULL);
   for (i = 0; i < nV; i++) {
      e[i] = malloc(nV * sizeof(int));
      assert(e[i] != NULL);
      for (j = 0; j < nV; j++)
         e[i][j] = FALSE;
   }
   Graph g = malloc(sizeof(GraphRep));
   assert(g != NULL);
   g->nV = nV;  g->nE = 0;  g->edges = e;
   return g;
}
// free memory allocated to graph
void  dropGraph(Graph g)
{
   assert(g != NULL);
   int i;
   for (i = 0; i < g->nV; i++)
      free(g->edges[i]);
   free(g->edges);
   free(g);
}
// show "printable" representation of graph
void  showGraph(Graph g)
```

```
{
   assert(g != NULL);
   printf("V=%d, E=%d\n", g->nV, g->nE);
   int i, j;
   for (i = 0; i < g->nV; i++) {
      int nshown = 0;
      for (j = i+1; j < g->nV; j++) {
         if (g->edges[i][j] != 0) {
            printf("%d-%d ",i,j);
            nshown++;
         }
      }
      if (nshown > 0) printf("\n");
   }
}
// add new edge to a graph
void insertE(Graph g, Edge e)
{
   assert(g != NULL);
   assert(validV(g,e.v) && validV(g,e.w));
   if (g->edges[e.v][e.w]) return;
   g->edges[e.v][e.w] = 1;
   g->edges[e.w][e.v] = 1;
   g->nE++;
}
// remove an edge from a graph
void removeE(Graph g, Edge e)
{
   assert(g != NULL);
   assert(validV(g,e.v) && validV(g,e.w));
   if (!g->edges[e.v][e.w]) return;
   g->edges[e.v][e.w] = 0;
   g->edges[e.w][e.v] = 0;
   g->nE--;
}
// returns #vertices & array of edges
int edges(Graph g, Edge *es, int nE)
{
   assert(g != NULL && es != NULL);
   assert(nE >= g->nE);
   int i, j, n = 0;
   for (i = 0; i < g->nV; i++) {
      for (j = i+1; j < g->nV; j++) {
         if (g->edges[i][j] != 0) {
            assert(n < nE);
            es[n++] = mkEdge(g,i,j);
         }
      }
   }
   return n;
}
```

Adjacency List Representation:

```
typedef struct vNode *VList;
struct vNode { Vertex v; vList next; };
typedef struct graphRep GraphRep;
struct graphRep {
   int   nV;     // #vertices
   int   nE;     // #edges
   VList *edges; // array of lists
};
// check whether two vertices are connected
```

```
int neighbours(Graph g, Vertex v, Vertex w)
{
   assert(validV(g,v) && validV(g,w));
   VList curr;
   curr = g->edges[v];
   while (curr != NULL) {
      if (curr->v == w) return 1;
   }
   return 0;
}
// make new graph with nV vertices
Graph newGraph(int nV)
{
   int i, j;
   VList *e = malloc(nV * sizeof(VList));
   assert(e != NULL);
   for (i = 0; i < nV; i++) e[i] = NULL;
   Graph g = malloc(sizeof(GraphRep));
   assert(g != NULL);
   g->nV = nV;  g->nE = 0;  g->edges = e;
   return g;
}
// free memory allocated to graph
void  dropGraph(Graph g)
{
   assert(g != NULL);
   int i;
   for (i = 0; i < g->nV; i++)
      freeVList(g->edges[i]);
   free(g);
}
// show "printable" representation of graph
void  showGraph(Graph)
{
   assert(g != NULL);
   printf("V=%d, E=%d\n", g->nV, g->nE);
   int i;
   for (i = 0; i < g->nV; i++) {
      vNode *n = g->edges[i];
      while (n != NULL) {
         printf("%d-%d ",i,n->v);
         n = n->next;
      }
      if (g->edges[i] != NULL) printf("\n");
   }
}
// add new edge to a graph
void  insertE(Graph g, Edge e)
{
   assert(g != NULL);
   assert(validV(g,e.v) && validV(g,e.w));
   int orig = length(g->edges[e.v]);
   g->edges[e.v] = insertVList(g->edges[e.v], e.w);
   g->edges[e.w] = insertVList(g->edges[e.w], e.v);
   if (length(g->edges[e.v]) > orig) g->nE++;
}
// remove an edge from a graph
void  removeE(Graph g, Edge e)
{
   assert(g != NULL);
   assert(validV(g,e.v) && validV(g,e.w));
   int orig = length(g->edges[e.v]);
   g->edges[e.v] = deleteVList(g->edges[e.v], e.w);
```

```
   g->edges[e.w] = deleteVList(g->edges[e.w], e.v);
   if (length(g->edges[e.v]) < orig) g->nE--;
}
// returns #vertices & array of edges
int edges(Graph g, Edge *es, int nE)
{
   VList curr;
   assert(g != NULL && es != NULL);
   assert(nE >= g->nE);
   int w, n = 0;
   for (w = 0; w < g->nV; w++) {
      curr = g->edges[w];
      while (curr != NULL) {
         if (w < curr->v)
            es[n++] = mkEdge(g,w,curr->v);
         curr = curr->next;
      }
   }
   return n;
}
```

Traversal

Path Checking:

```
int *visited;  // array of booleans
               // indexed by vertex 0..V-1

// DFS : depth-first search
int hasPath(Graph g, Vertex src, Vertex dest)
{
   int i;
   visited = malloc(g->nV*sizeof(int));
   for (i = 0; i < g->nV; i++) visited[i] = 0;
   return dfsPathCheck(g, src, dest);
}
int dfsPathCheck(Graph g, Vertex v, Vertex dest)
{
   visited[v] = 1;
   Vertex w;
   for (w = 0; w < g->nV; w++) {
      if (g->edges[v][w] && w == dest)
         return 1; // found path
      if (g->edges[v][w] && !visited[w])
         return dfsPathCheck(g, w);
   }
   return 0; // no path from src to dest
}

// BFS : breadth-first search
int hasPath(Graph g, Vertex src, Vertex dest)
{
   int *visited = calloc(g->nV,sizeof(int));
   Queue q = newQueue();
   QueueJoin(q,src);
   int isFound = 0;
   while (!QueueIsEmpty(q) && !isFound) {
      Vertex y, x = QueueLeave(q);
      if (visited[x]) continue;
      for (y = 0; y < g->nV; y++) {
         if (!g->edges[x][y]) continue;
         if (y == dest) { isFound = 1; break; }
         if (!visited[y]) { QueueJoin(q,y); }
```

```
      }
   }
   free(visited);
   return isFound;
}
```

Find and Display Shortest Path:

```
int findPath(Graph g, Vertex src, Vertex dest)
{
   int i;
   // array of "been visited" flags
   int *visited = malloc(g->nV * sizeof(int));
   for (i = 0; i < g->nV; i++) visited[i] = 0;
   // array of path predecessor vertices
   Vertex *path = malloc(g->nV * sizeof(Vertex));
   Queue q = newQueue();
   QueueJoin(q, src); visited[src] = 1;
   int isFound = 0;
   while (!emptyQ(q) && !isFound) {
      Vertex y, x = QueueLeave(q);
      for (y = 0; y < g->nV; y++) {
         if (!g->edges[x][y]) continue;
         path[y] = x;
         if (y == dest) { isFound = 1; break; }
         if (!visited[y]) {
            QueueJoin(q, y);
            visited[y] = 1;
         }
      }
   }
   if (isFound) {
      // display path in dest..src order
      Vertex v;
      for (v = dest; v != src; v = path[v])
          printf("%d<-", v);
      printf("%d\n", src);
   }
}
```

Connected Components:

```
int *componentOf;  // array of component ids
                   // indexed by vertex 0..V-1
int ncounted;      // # vertices included so far

void components(Graph g)
{
   void dfsComponents(Graph,Vertex,int);
   int i, comp = 0;
   componentOf = malloc(g->nV*sizeof(int));
   for (i = 0; i < g->nV; i++) componentOf[i] = -1;
   ncounted = 0;
   while (ncounted < g->nV) {
      Vertex v;
      for (v = 0; v < g->nV; v++)
         if (componentOf[v] == -1) break;
      dfsComponents(g, v, comp);
      comp++;
   }
   // componentOf[] is now set
}
void dfsComponents(Graph g, Vertex v, int c)
```

```
{
   componentOf[v] = c;
   ncounted++;
   Vertex w;
   for (w = 0; w < g->nV; w++) {
      if (g->edges[v][w] && componentOf[w] == -1)
         dfsComponents(g, w, c);
   }
}
```

Hamilton Path Check:

```
int *visited; // array of nV bools
int HamiltonR(Graph g, Vertex v, Vertex w, int d)
{
   int t;
   if (v == w) return (d == 0) ? 1 : 0;
   visited[v] = 1;
   for (t = 0; t < g->nV; t++) {
      if (!neighbours(g,v,t)) continue;
      if (visited[v] == 1) continue;
      if (HamiltonR(g,t,w,d-1)) return 1;
   }
   visited[v] = 0;
   return 0;
}
int hasHamiltonPath(Graph g, Vertex src, Vertex dest)
{
   visited = calloc(g->nV,sizeof(int));
   int res = HamiltonR(g, src, dest, g->nV-1);
   free(visited);
   return res;
}
```

Digraphs

Digraph Interface/Representation:

```
// Same interface as for simple graphs
// Representation changes:
// - essentially the same data structures
// - if using adjacency matrix, no longer symmetric
// - if using adjacency lists, no longer store (v,w) and (w,v)
```

Web Crawler:

```
// abstract algorithm only
webCrawler(startingURL)
{
   mark startingURL as alreadySeen
   enqueue(Q, startingURL)
   while not empty(Q) {
      nextPage = dequeue(Q)
      visit nextPage
      foreach (hyperLink in nextPage) {
         if (hyperLink not alreadySeen) {
            mark hyperLink as alreadySeen
            enqueue(Q, hyperLink)
         }
      }
   }
}
```

Reachability, Transitive Closure:

```
int **tc; // VxV matrix indicating reachability

// build an empty VxV matrix
int **makeMatrix(int nrows, int ncols, int init)
{
   int i, j;
   int **m = malloc(nrows * sizeof(int *));
   assert(m != NULL);
   for (i = 0; i < nrows; i++) {
      m[i] = malloc(ncols*sizeof(int));
      assert(m[i] != NULL);
      for (j = 0; j < ncols; j++) m[i][j] = init;
   }
   return m;
}
// build transitive closure matrix
void makeClosure(Graph g)
{
   int i, s, t, V = g->nV;
   tc = makeMatrix(V, V, 0);
   for (s = 0; s < V; s++) {
      for (t = 0; t < V; t++)
         tc[s][t] = g->edges[s][t];
   }
   for (i = 0; i < V; i++) {
      for (s = 0; s < V; s++) {
         if (tc[s][i] == 0) continue;
         for (t = 0; t < V; t++)
            if (tc[i][t] == 1) tc[s][t] = 1;
      }
   }
   g->tc = tc;
}
// is there some path from src to dest
int reachable(Graph g, Vertex src, Vertex dest)
{
    if (g->tc == NULL) makeClosure(g);
    return g->tc[src][dest];
}
```

Weighted Graphs

Graph Interface:

```
// visible data structures for Graphs
typedef struct GraphRep *Graph;
// vertices denoted by integers 0..N-1
typedef int Vertex;
// edges are end-points + weight
typedef struct { Vertex src; Vertex dest; float weight; } Edge;
// auxiliary operations on graphs
int validV(Graph,Vertex); // validity check
Edge mkEdge(Graph, Vertex, Vertex, float); // edge creation
int neighbours(Graph, Vertex, Vertex); // edge existence
float compareE(Edge e1, Edge e2); // compare edge weights
// core operations on graphs
// make new graph with nV vertices
Graph newGraph(int nV);
// free memory allocated to graph
void  dropGraph(Graph);
```

```
// show "printable" representation of graph
void  showGraph(Graph);
// add new edge to a graph
void  insertE(Graph, Edge);
// remove an edge from a graph
void  removeE(Graph, Edge);
// returns #vertices & array of edges
int   edges(Graph, Edge *, int);
```

Implementation of Auxiliary Operations:

```
// is a vertex valid in a given Graph?
static int validV(Graph g, Vertex v)
{
   return (g != NULL && v >= 0 && v < g->nV);
}
// make an Edge value
Edge mkEdge(Graph g, Vertex v, Vertex w)
{
   assert(validV(g,v) && validV(g,w));
   Edge new;
   new.src = src;
   new.dest = dest;
   new.weight = weight;
   return new;
}
// compare Edge weights
int compareE(Edge e1, Edge e2)
{
   return e1.weight - e2.weight;
}
```

Adjacency Matrix Representation:

```
// since 0 is a valid weight, can't use it for "no edge"
// need a distinguished value to indicate "no edge"
#define NO_EDGE MAXFLOAT  // imaginary distinguished float value
typedef struct GraphRep {
   int    nV;    // #vertices
   int    nE;    // #edges
   Bool **edges; // matrix of booleans
} GraphRep;
// check whether two vertices are connected
int neighbours(Graph g, Vertex v, Vertex w)
{
        assert(validV(g,v) && validV(g,w));
        return (g->edges[v][w] != NO_EDGE);
}
// make new graph with nV vertices
Graph newGraph(int nV)
{
   assert(nV >= 0);
   int i, j;
   float **e = malloc(nV * sizeof(float *));
   assert(e != NULL);
   for (i = 0; i < nV; i++) {
      e[i] = malloc(nV * sizeof(float));
      assert(e[i] != NULL);
      for (j = 0; j < nV; j++)
         e[i][j] = NO_EDGE;
   }
   Graph g = malloc(sizeof(GraphRep));
   assert(g != NULL);
```

```
   g->nV = nV;  g->nE = 0;  g->edges = e;
   return g;
}
// free memory allocated to graph
void  dropGraph(Graph g)
{
   assert(g != NULL);
   int i;
   for (i = 0; i < g->nV; i++)
      free(g->edges[i]);
   free(g->edges);
   free(g);
}
// show "printable" representation of graph
void  showGraph(Graph g)
{
   assert(g != NULL);
   printf("V=%d, E=%d\n", g->nV, g->nE);
   int i, j;
   for (i = 0; i < g->nV; i++) {
      int nshown = 0;
      for (j = i+1; j < g->nV; j++) {
         float wt = g->edges[i][j];
         if (wt != NO_EDGE) {
            printf("%d-%0.1f-%d ",i,wt,j);
            nshown++;
         }
      }
      if (nshown > 0) printf("\n");
   }
}
// add new edge to a graph
void insertE(Graph g, Edge e)
{
   assert(g != NULL);
   Vertex v = e.src, w = e.dest;
   assert(validV(g,v) && validV(g,w));
   if (G->edges[v][w] == NO_EDGE) g->nE++;
   g->edges[v][w] = e.weight;
}
// remove an edge from a graph
void removeE(Graph g, Edge e)
{
   assert(g != NULL);
   Vertex v = e.src, w = e.dest;
   assert(validV(g,v) && validV(g,w));
   if (g->edges[v][w] == NO_EDGE) return;
   g->edges[v][w] = NO_EDGE;
   g->edges[w][v] = NO_EDGE;
   g->nE--;
}
// returns #vertices & array of edges
int edges(Graph g, Edge *es, int nE)
{
   assert(g != NULL && es != NULL);
   assert(nE >= g->nE);
   int i, j, n = 0;
   for (i = 0; i < g->nV; i++) {
      for (j = i+1; j < g->nV; j++) {
         if (g->edges[i][j] != NO_EDGE) {
            assert(n < nE);
            es[n++] = mkEdge(g,i,j);
         }
```

```
      }
   }
   return n;
}
```

Minimum Spanning Tree (Kruskal):

```
typedef Graph MSTree; // an MST is a specialised Graph
// assumes existence of list-of-edges ADT
MSTree kruskalFindMST(Graph g)
{
   Graph mst = newGraph(); //MST initially empty
   EdgeList eList; //sorted list of edges
   int i;  Edge e;  int eSize = sizeof(Edge);
   edges(eList, g->nE, g);
   eList = qsort(sorted, g->nE, eSize, compareE);
   for (i = 0; mst->nE < g->nV-1; i++) {
      e = eList[i];
      insertE(mst, e);
      if (hasCycle(mst)) removeE(mst, e);
   }
}
```

Minimum Spanning Tree (Prim):

```
typedef Graph MSTree; // an MST is a specialised Graph
// assumes existence of set-of-edges ADT
// assumes existence of set-of-vertices ADT
MSTree primFindMST(Graph g)
{
   EdgeSet mst = {}; //MST initially empty
   VertexSet vSet = {0}; // start vertex
   EdgeSet fringe = {}; //edges at "fringe"
   Vertex curr, s, t;  Edge e;  float w;
   fringe = edgesAt(0);
   while (card(vSet) < g->nV) {
      find e in fringe with minimum cost
      fringe = exclude(fringe, e)
      (s,curr,w) = e
      vSet = include(vSet, curr)
      mst = include(mst, e)
      foreach (e in edgesAt(curr)) {
         (s,t,w) = e  // s == curr
         if (!isElem(t,vSet))
            fringe = include(fringe,e)
      }
   }
}
```

Single-source Shortest Path (Dijkstra):

```
float *dist; dist[d] = distance of shortest path from s..d
Vertex *pred; pred[v] = predecessor of v in shortest path
// assumes existence of set-of-vertices ADT
// assumes existence of priority queue ADT
// abstract algorithm only ...
MSTree shortestPath(Graph g, Vertex s)
{
   VertexSet vSet = {}; // visited vertices
   PQueue todo = newPQueue(); // edges to be considered
   float *dist = malloc(g->nV*sizeof(float);
   for (int i = 0; i < g->nV; i++) dist[i] = MAXFLOAT;
```

```
   dist[s] = 0.0;
   Vertex *pred = malloc(g->nV*sizeof(Vertex));
   for (i = 0; i < g->nV; i++) pred[i] = -1;
   Vertex v, w, s, t;  Edge e;  float wt;
   PQueueJoin(todo, s, dist[s]);
   while (!PQueueIsEmpty(todo)) {
      v = PQueueLeave(todo);
      if (isElem(vSet,v)) continue;
      vSet = include(vSet, v)
      foreach (e=(v,w,wt) in edgesAt(v)) {
         if (dist[v] + wt < dist[w]) {
            dist[w] = dist[v] + wt;
            pred[w] = v;
            PQueueJoin(todo, w, dist[w]);
         }
      }
   }
}
```