

List ADT operations (pa6):

```
List();
List(const List& L);
~List();
int length() const;
ListElement front() const;
ListElement back() const;
int position() const;
ListElement peekNext() const;
ListElement peekPrev() const;
void clear();
void moveFront();
void moveBack();
ListElement moveNext();
ListElement movePrev();
void insertAfter(ListElement x);
void insertBefore(ListElement x);
void setAfter(ListElement x);
void setBefore(ListElement x);
void eraseAfter();
void eraseBefore();
int findNext(ListElement x);
int findPrev(ListElement x);
void cleanup();
List concat(const List& L) const;
std::string to_string() const;
bool equals(const List& R) const;
friend std::ostream& operator<<( std::ostream& stream, const List& L );
friend bool operator==( const List& A, const List& B );
List& operator=( const List& L );
```

Graph ADT operations (pa2):

```
Graph newGraph(int n);
void freeGraph(Graph* pG);
int getOrder(Graph G);
int getNumEdges(Graph G);
int getNumArcs(Graph G);
int getSource(Graph G);
int getParent(Graph G, int u);
int getDist(Graph G, int u);
void getPath(List L, Graph G, int u);
void makeNull(Graph G);
void addEdge(Graph G, int u, int v);
void addArc(Graph G, int u, int v);
void BFS(Graph G, int s);
void printGraph(FILE* out, Graph G);
```

Graph ADT operations (pa3):

```
Graph newGraph(int n);
void freeGraph(Graph* pG);
int getOrder(Graph G);
int getNumEdges(Graph G);
int getNumArcs(Graph G);
int getParent(Graph G, int u);
int getDiscover(Graph G, int u);
int getFinish(Graph G, int u);
void makeNull(Graph G);
void addEdge(Graph G, int u, int v);
void addArc(Graph G, int u, int v);
void DFS(Graph G, List S);
Graph copyGraph(Graph G);
Graph transpose(Graph G);
void printGraph(FILE* out, Graph G);
```

Some Graph Algorithms

Initialize(G, s)

```
for all x in V(G)
    d[x] = inf
    p[x] = nil
d[s] = 0
```

Relax(x, y) pre: y in adj[x]

```
if d[y] > d[x] + w(x,y)
    d[y] = d[x] + w(x,y)
    p[y] = x
```

Dijkstra(G, s) pre: all edge weights non-negative

```
Initialize(G, s)
S = { }    // the set of discovered vertices
Q = V(G)   // create a Min Priority Queue with key[x] = d[x]
while Q is not empty
    x = ExtractMin(Q)
    S = S union {x}
    for all y in adj[x]
        Relax(x,y)
```

Some Heap Algorithms

Heapify(A, i) pre: subtrees at left(i) and right(i) are valid heaps

```
l = left(i)
r = right(i)
if l <= heapSize[A] and A[l] > A[i]
    largest = l
else
    largest = i
if r <= heapSize[A] and A[r] > A[largest]
    largest = r
if largest != i
    A[i] <--> A[largest]    // swap
    Heapify(A, largest)
```

BuildHeap(A)

```
n = heapSize[A] = length[A]
for i=floor(n/2) down to 1
    Heapify(A, i)
```

HeapSort(A)

```
BuildHeap(A)
n = length[A]
for i=n down to 2
    A[1] <--> A[i]    // swap
    heapSize[A]--
    Heapify(A, 1)
```

Some RBT Algorithms:

LeftRotate(T, x)

```
y = x.right
x.right = y.left
if y.left != T.nil
    y.left.parent = x
y.parent = x.parent
if x.parent == T.nil
    T.root = y
else if x == x.parent.left
    x.parent.left = y
else
    x.parent.right = y
y.left = x
x.parent = y
```

RightRotate(T, x)

```
y = x.left
x.left = y.right
if y.right != T.nil
    y.right.parent = x
y.parent = x.parent
if x.parent == T.nil
    T.root = y
else if x == x.parent.right
    x.parent.right = y
else
    x.parent.left = y
y.right = x
x.parent = y
```

RB Insert(T, z)

```
y = T.nil
x = T.root
while x != T.nil
    y = x
    if z.key < x.key
        x = x.left
    else
        x = x.right
z.parent = y
if y == T.nil
    T.root = z
else if z.key < y.key
    y.left = z
else
    y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB_InsertFixUp(T, z)
```

```

RB InsertFixUp(T, z)
while z.parent.color == RED
    if z.parent == z.parent.parent.left
        y = z.parent.parent.right
        if y.color == RED
            z.parent.color = BLACK           // case 1
            y.color = BLACK                 // case 1
            z.parent.parent.color = RED     // case 1
            z = z.parent.parent            // case 1
        else
            if z == z.parent.right
                z = z.parent                // case 2
                LeftRotate(T, z)            // case 2
                z.parent.color = BLACK      // case 3
                z.parent.parent.color = RED // case 3
                RightRotate(T, z.parent.parent) // case 3
            else
                y = z.parent.parent.left
                if y.color == RED
                    z.parent.color = BLACK // case 4
                    y.color = BLACK        // case 4
                    z.parent.parent.color = RED // case 4
                    z = z.parent.parent    // case 4
                else
                    if z == z.parent.left
                        z = z.parent        // case 5
                        RightRotate(T, z)   // case 5
                        z.parent.color = BLACK // case 6
                        z.parent.parent.color = RED // case 6
                        LeftRotate(T, z.parent.parent) // case 6
T.root.color = BLACK

```