CS 261 - Spring 2011

List Stack
List Bag
List Queue

by
Tim Budd
Ron Metoyer
Sinisa Todorovic
Dynamic Array -- Problems

• Data kept in a single large block of memory

• Often more memory used than necessary
  – especially when repeatedly growing and shrinking the dynamic array
Linked List

• A good alternative

• The memory use is always proportional to the number of elements in the collection
Characteristics of Linked Lists

- Elements are held in objects called **Links**
- Links are 1-1 with elements, allocated and released as necessary
- Each link points to next link in sequence, sometimes to previous link
Link Structure

```c
struct link {
    TYPE    value;
    struct link * next;
};
```
Elements of Linked Lists

- **Header** -- special link for start
- Use null as terminator
- **Sentinel** -- special link for end
- Use single or double links?
- Pointer to first element only
- Pointer to first and last element
Example: List Stack

```
firstLink =

val: 2
next: 

val: 7
next: 

val: 4
next: null
```
Example: Add Stack

```
firstLink =

val: 2
next: val: 7
next: val: 4
next: null
```
Characteristics of Linked Lists

- Elements are held in objects called **Links**
- Links are 1-1 with elements, allocated and released as necessary
- Each link points to next link in sequence, sometimes to previous link
Link Structure

```c
struct link {
    TYPE value;
    struct link * next;
};
```
List Stack

- Keep pointer to first element
- Use null pointer if stack empty
- Add or remove elements only from front
- Allow only singly linked list
- Can access only first element
Implementation of List Stack

```c
struct link {
    TYPE value;
    struct link * next;
};

struct listStack {
    struct link * firstLink;
};
```

How to initialize List Stack?
Push List Stack: 3 Steps

allocate new link

firstLink =

val: 5
next: null

val: 2
next:

val: 7
next:

val: 4
next: null
Push List Stack: 3 Steps

- firstLink =
- add the element to top
- val: 5
  next: val: 2
  next: val: 7
  next: val: 4
  next: null
- link the element
Push List Stack: 3 Steps

```c
void pushStack (struct listStack *stk, TYPE val)
{
    stk->firstLink = _newLink(val, stk->firstLink);
}
```
Internal utility routine

```c
struct link * _newLink (TYPE v, struct link * old)
{
    struct link * new =
        (struct link *) malloc(sizeof(struct link));
    assert (new != 0);
    new->value = v;
    new->next = old;
    return new;
}
```
topStack, pushStack, isEmpty...

- Should be done on your own
- Worksheet 17
# List Stack vs. Dyn. Array Stack

<table>
<thead>
<tr>
<th>Function</th>
<th>List</th>
<th>Dyn. Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushStack</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>popStack</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>topStack</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
List Bag

struct link {
    TYPE value;
    struct link * next;
};

struct listBag {
    struct link * firstLink;
}
List Bag

- Init, Add operations are similar to List Stack
- Contains and Remove operations are tricky
- How to patch up links after removing an element?
void removeListBag(struct listBag *b, TYPE val) {
    struct link *previous = b->firstLink;
    struct link *current;
    for (current = previous->next; current != 0; 
        current = current->next)
    {
        if (EQ(current->value, val)) {
            // do what needs to be done
        }
        previous = current;
    }
}

When you find it

- When you find the element to be deleted, what does previous point to?

- What if the element to be deleted is at the front of the list? Does this matter?
Array Queue vs. List Queue

• An array queue has larger complexity

• Adding an element to the beginning, must move all elements => O(n)
Array Queue vs. List Queue

• With lists just keep a pointer to both the front AND the back

• Elements added to the back, removed from front
Example of List Queue

```
List

head

link

link

... 

link

```
Why add to back, remove from front?

- Think about the issues
- Why not the other way around?
Special Cases

• Note for popStack we needed to treat removing last element as a special case

• Need to check for removing last val
Sentinel

- Helps remove special cases due to null references since it’s never null
- Simplifies some operations
Sentinel

- A sentinel is a special marker at the front and/or back of the list
- Has no value
- Never removed
Implementation of ListQueue

Worksheet 18
Structure for List Queue

```c
struct link {
    TYPE    value;
    struct link * next;
};

struct listQueue {
    struct link * head;
    struct link * tail;
};
```
void initListQueue (struct listQueue *q){

    struct link *lnk =
    (struct link *)malloc(sizeof(struct link));

    assert(lnk != 0); /* lnk is the sentinel */

    lnk->next = 0;

    q->head = q->tail = lnk;
}

isEmpty

int isEmptyListQueue (struct listQueue *q) {

}

Elements are Added to Tail

```c
void addBackListQueue (struct listQueue *q, TYPE val)
{
    struct link * lnk;
    lnk->next = 0;
    lnk->value = val;
    q->tail->next = lnk;
    q->tail = lnk;
}
```
void addBackListQueue (struct listQueue *q, TYPE val)
{
    struct link * lnk;
    lnk->next = 0;
    lnk->value = val;
    q->tail->next = lnk;
    q->tail = lnk;
}
Elements are Added to Tail

```c
void addBackListQueue (struct listQueue *q, TYPE val)
{
    struct link * lnk;
    lnk->next = 0;
    lnk->value = val;
    q->tail->next = lnk;
    q->tail = lnk;
}
```
Add to Tail -- No Sentinel

```c
void addBackListQueue (struct listQueue *q, TYPE val) {
    struct link * lnk;
    lnk->next = 0;
    lnk->value = val;

    /* Special case for when the queue is empty */
    if(!isEmptyListQueue(q)){
        q->tail->next = lnk;
        q->tail = lnk;
    }else q->head = q->tail = lnk;
}
```
Remove from Front

```c
void removeFromListQueue (struct listQueue *q) {
    struct link * lnk = q->head->next;
    assert ( ! isEmptyListQueue(q));
    q->head->next = lnk->next;
    if(q->head->next == 0) q->tail = q->head;
    free (lnk);
}
```
Remove from Front

```c
void removeFromListQueue (struct listQueue *q) {
    struct link * lnk = q->head->next;
    assert ( ! isEmptyListQueue(q));
    q->head->next = lnk->next;
    if(q->head->next == 0) q->tail = q->head;
    free (lnk);
}
```
void removeFromListQueue (struct listQueue *q) {
    struct link * lnk = q->head->next;
    assert ( ! isEmptyListQueue(q));
    q->head->next = lnk->next;
    if(q->head->next == 0) q->tail = q->head;
    free (lnk);
}
Remove from Front

```c
void removeFromListQueue (struct listQueue *q) {
    struct link * lnk = q->head->next;
    assert ( ! isEmptyListQueue(q));
    q->head->next = lnk->next;
    if(q->head->next == 0) q->tail = q->head;
    free (lnk);
}
```
Worksheets 17, 17b

• List Stack (should be trivial)

• List Queue (only slightly harder)

• List Bag (requires a little more thought)