Introduction to Linked Lists

• In your previous programming course, you organized and processed data items sequentially using an array (or possibly an arraylist, or a vector).

• You probably performed several operations on arrays, such as sorting, inserting, deleting, and searching.

• If data items are not sorted, then searching for an item in the array can be very time-consuming, especially with large arrays.
Introduction to Linked Lists

• Once the data items are sorted, we can use a binary search and improve the search algorithm; however, insertion and deletion become time-consuming, especially with large arrays, because these operations require data movement.

• With an array of fixed size, new items can be added only if there is room. Thus, there are limitations when organizing data in an array.
Introduction to Linked Lists

• One way to improve performance when there is a need to insert and/or delete items is to use a data structure that is not contiguous in memory, such as a linked list.
Linked Lists

• a data structure in which each element is dynamically allocated and in which elements point to each other to define a linear relationship.

• elements are allocated separately and only when needed

• elements are usually the same type (but not always)
Linked Lists

- each element of the structure is typically called a **node**

*Example of a singly-linked list with a head pointer. Nodes are added to the **front** of the list.*

- - indicates that the pointer is NULL
Introduction to Linked Lists

Example of a singly-linked list with a head and tail pointer. Nodes can be added to the front or the end of the list.
Comparing Arrays and Linked Lists

Both arrays and linked lists can be used to store linear data of similar types, but they both have some advantages and disadvantages over each other.

Advantages of Arrays:

• We can randomly access an element of an array by using its index.

• When sorted, we can use a binary search to improve the searching algorithm
Comparing Arrays and Linked Lists

Disadvantage of Arrays:

• The size of an array is fixed: So we must know the upper limit on the number of elements in advance.
  – Generally, the allocated memory is equal to the upper limit irrespective of the usage
  – In practical uses, upper limit is rarely reached; therefore wasted memory.

• With an array of fixed size, new items can be added only if there is room.
Comparing Arrays and Linked Lists

Disadvantage of Arrays:

• Inserting new elements is time-consuming, because room has to be created for the new elements and, to create room, existing elements have to be shifted (see zyBooks 7.2 regarding vectors).

• Deleting elements is time-consuming, if deleting elements not at the end of the array. e.g., to delete the $i^{th}$ element, everything after the $i^{th}$ element must be shifted up.
Comparing Arrays and Linked Lists

Advantages of Linked Lists:

• We can increase the size of the linked list one item at a time, dynamically.
• We can easily insert elements in a linked list.
• We can easily delete elements from a linked list.
Comparing Arrays and Linked Lists

Disadvantages of Linked Lists:

• Random access is not allowed. We have to access elements sequentially, starting from the first node. So we cannot do binary search with linked lists.

• Require extra memory space for a pointer in each node of the list.
Introduction to Linked Lists

Types of Linked Lists

- A singly linked list as previously described
- A doubly linked list - a list that has two references, one to the next node and another to the previous node.
Singly Linked List

typedef struct listItemType
{
    type payload;
    struct listItemType *next;
}listItem;
listItem *head;
Adding an Item to a List

```c
listItem *p, *q;
```

- Add an item pointed to by `q` after item pointed to by `p`
  - Neither `p` nor `q` is `NULL`
Adding an Item to a List

```c
listItem *addAfter(listItem *p, listItem *q) {
    q -> next = p -> next;
    p -> next = q;
    return p;
}
```
Adding an Item to a List

```c
listItem *addAfter(listItem *p, listItem *q){
    q -> next = p -> next;
    p -> next = q;
    return p;
}
```
Adding an Item to a List

```c
listItem *addAfter(listItem *p, listItem *q) {
    q -> next = p -> next;
    p -> next = q;
    return p;
}
```

Question: What should we do if we cannot guarantee that \( p \) and \( q \) are non-NULL?
Adding an Item to a List (continued)

```c
listItem *addAfter(listItem *p, listItem *q) {
    if (p && q) {
        q -> next = p -> next;
        p -> next = q;
    }
    return p;
}
```

Note: test for non-null p and q
What about Adding an Item \textit{before} another Item?

\textbf{struct listItem *p;}

- Add an item \textit{before} item pointed to by $p$ ($p \neq$ NULL)
What about Adding an Item *before* another Item?

- **Answer:**
  - Need to search list from beginning to find previous item
  - Add new item after previous item
Linked Lists (continued)

• We want to develop a list management system that allows us to maintain lists of any type of data (integers, floats, structures, etc.).

• We will use \textit{void} pointers to accomplish this.
A review of void Pointers and type casting

- A **void** pointer (i.e., **void**) is a generic pointer that can hold the address of any pointer type. Consider the following segment of code on the right:

  ```c
  int x;
  float y;
  typedef struct dataType
  {
    int f1;
    int f2;
  } data_t;
  data_t d1;
  data_t *dptr;
  void *ptr;

  ptr = &x;
  ptr = &y;
  ptr = &d1;
  ptr = dptr;
  dptr = ptr;
  ```

- The **void** pointer `ptr` can be set to point to any of the objects, and any type pointer can be set to `ptr`
A review of void Pointers and type casting

• Although a *void can be set to point to any type, you should not assign a pointer to a given type to a pointer to another type.

• Consider the code on the right

```
int x = 55;
float y = 83.7;
int *iPtr;
int *fPtr = &y;
iPtr = fPtr;
```

Note that iPtr is a pointer to an int and y is a float

You will get the following message: “...warning: assignment from incompatible pointer type” and generally unexpected behavior.
A review of void Pointers and type casting

• You can never dereference a void pointer.
• i.e., `void *` pointers can never be directly used to access the memory to which they point because the size and type of the location are unknown.

Example:
```c
int x;
float y;
typedef struct dataType {
    int f1;
    int f2;
} data_t;
data_t d1;
data_t *dptr;
void *ptr;
...
x = *ptr;
...
```

...warning: dereferencing 'void *' pointer
...error: invalid use of void expression
A review of void Pointers and type casting

There are two ways to avoid this problem:

1. type casting: You can “tell” the compiler the type of the data that “ptr” is pointing to, i.e.:

   ```
   x = *(int *)ptr;
   ```

   This is telling the compiler to treat `ptr` as a pointer to something of type “int”, and then copy the integer contents it is "pointing to" to `x`.

2. Define a separate integer pointer and copy `ptr` to it; the other technique is to simply define another pointer, i.e.:

   ```
   int *intPtr;
   intPtr = ptr;
   x = *intPtr;
   ```
A review of void Pointers and type casting

• Defining a few extra pointers of the appropriate type may result in more readable code and will likely be less error-prone (but either approach is okay).

• Here is another example using the structure on the right. Assume we want to set the “f2” field in “d1”.

• Note: ptr is pointing to d1

Example:

```c
int x;
float y;
typedef struct data_type
{
    int f1;
    int f2;
} data_t;
data_t d1;
data_t *dptr;
void *ptr = &d1;

/* set f2 field in d1   */
((data_t *)ptr)->f2 = 6;

/* or   */
dptr = ptr;
dptr->f2 = 6;
```
Linked List (continued)

• A linked list is a very useful data structure and many of the techniques used to manipulate linked lists are applicable to other data structures.

• In lab, you will develop a list management module.
List data structures

The node_t structure

• The following example creates a new type name, node_t, which is 100% equivalent to struct node_type

```c
/** List node **/
typedef struct node_type
{
    struct node_type *next; /* Pointer to next node */
    void *dataPtr; /* Pointer to associated object */
} node_t;
```

• There is a single instance of the node_t structure for each element in each list.

• **Note:** we must use the "official" name struct node_type when declaring next because node_t is not known to be a type definition until 3 lines later!

• Each node contains two pointers:
  – One is to the next node_t in the list.
  – The other points to the actual data being managed by the list
List data structures

The node_t structure

/** List node **/
typedef struct node_type
{
    struct node_type *next; /* Pointer to next node */
    void *dataPtr; /* Pointer to associated object */
} node_t;

• The dataPtr pointer is declared to be of type void *. As stated earlier,
  – A void * pointer is a pointer to something of unknown or generic type.
  – void * pointers can be freely assigned to other pointers
  – void * pointers can never be directly used to access the memory
    to which they point because the size and type of the location are
    unknown.
List data structures

/** List structures **/
typedef struct list_type
{
    node_t *head; /* Pointer to front of list */
} list_t;

• There is a single instance of the list_t structure for each list.
list_t Functions

newList() – a function used to create a new list. In a true O-O language, each class has a constructor method that is automatically invoked when a new instance of the class is created. The newList() function serves this role here. Its mission is to:

1 - malloc() a new list_t structure.
2 - set the head pointer of the structure to NULL.
3 - return a pointer to the list_t to the caller.

list_t *newList()
{
}

Functions

`list_add()` must add the element pointed to by `objPtr` to the list structure pointed to by `list`. Its mission is to:

1. `malloc()` a new instance of `node_t`,
2. assign `dataPtr` to the `dataPtr` of the new `node_t`
3. add the new `node_t` to the front of the list, i.e.
   a. assign the `next` pointer of the new node to point to what `head` is pointing to, and
   b. assign `head` to point to the new `node_t`

```c
void list_add(list_t *list, void *dataPtr) {
}
```
Processing a list – the iterator_t object

• The list_t functions described above provide a means to add nodes to the list, but there are no functions to process the nodes of a list (i.e. retrieve the nodes).

• Both Java and C++ provide another class known as the iterator class that is associated with a list and can be used to sequentially retrieve elements of a list.

• We will develop our own version of an iterator.
/** Iterator **/

typedef struct list_iterator
{
    list_t *list; /* List iterator is associated with */
    node_t *position; /* Current position in list */
} iterator_t;

The “list” field points to the list_t the iterator is associated with, and “position” points to the node that would be retrieved next.
list_t Functions

iterator_t *newIterator(list_t *list);

list_reset() – sets the position pointer to the head pointer. If the list is empty this will cause the position pointer to be set to NULL.

/* Reset position to front of list */
void list_reset(iterator_t *iter)
{
}

list_t Functions

list_hasnext() – tests for the end of the list. Returns 1 if not at the end of the list; otherwise returns 0.

```
int list_hasnext(iterator_t *iter)
{
}
```

The list_hasnext() function should be called BEFORE list_next() is invoked to make sure the current pointer points to a valid node!
list_t Functions

`list_next()` — returns the address of the data pointed to by the `node` to which the `position pointer` points. The function should advance the `position` pointer so that it points to the next node in the list. If the `position` pointer is presently pointing at the last node in the list, then this call will and should set the current pointer to NULL.

```c
void *list_next(iterator_t *iter)
{
}
```