

INTRODUCTION

In this week we cover the following topics

6.1 Lists

6.1.1 Singly-linked lists

6.1.2 Doubly-linked lists

6.2 Friend Classes

6.3 Iterators

6.4 `dlist.h`

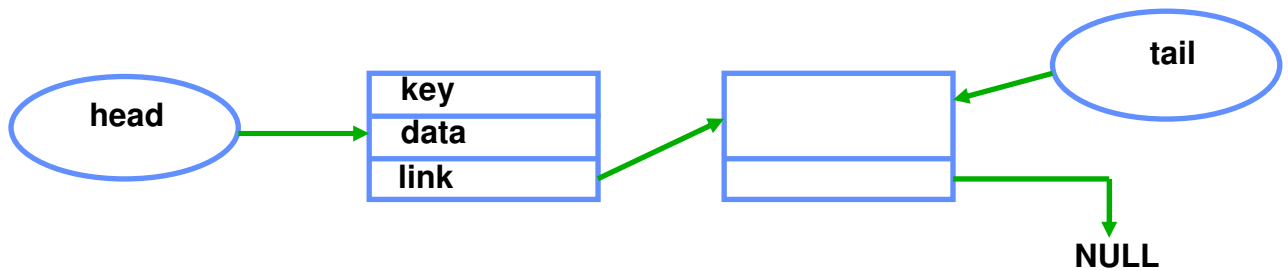
6.1 LISTS

- A list is a set of structures, each containing
 - One or more pointers to act as *links*.
 - At least one data item to act as a *key* for searching or sorting the list.
 - Any other data.
- We also need a pointer to the head of the list and it may be useful to have a pointer to the tail of the list.
- This is an application that combines classes and pointers. While not as fast as vectors they are very flexible in the way they can grow and shrink. Lists and their variants are “natural” data structures in that the concepts behind them are the basis for more sophisticated data structures.
- Basic problems:
 - Maintaining linkages
 - Adding and removing items
 - Finding items in the list.Mechanisms to achieve this range from simple to complex.

6.1.1 Singly-Linked Lists

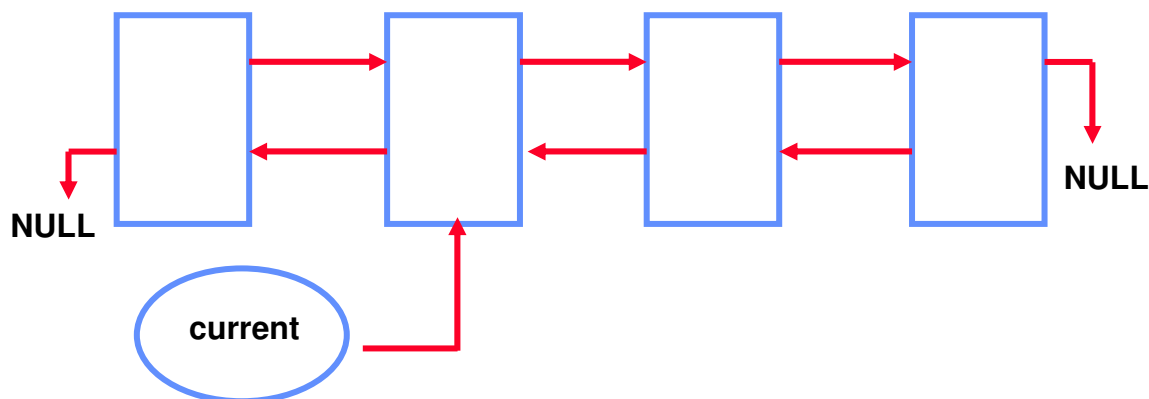
- Each item points to the next.
- Tail pointer is optional, makes it easier to add to tail.
- Items can be added or removed anywhere in list
 - (But you have to search along the list to find insertion point)
- Last item's pointer is `NULL`, signifies end of list

- If `head` and `tail` pointers are `NULL`, list is empty



6.1.2 Doubly-Linked Lists

- Each node contains two pointers
 - One points to right
 - The other points left
- Because we can walk the list in both directions, only one pointer is needed to manage the list.



- Some operations are simpler.
- Some operations are faster.
- Most operations are more complex.
- Code is harder to understand, harder to debug.

6.2 FRIEND CLASSES

- There are times when you want a class to be able to directly access the private data and functions of another class.

- This can be achieved by making one class a *friend* of another. The friend class is able to directly access the private data.
- The following example illustrates.

```
class xyz
{
    friend abc;

    private:
        int x;
};

class abc
{
    private:
        int a;

    public:
        void seta(xyz &foo) {
            a = foo.x;
        }
}
```

If `abc` was not a friend of `xyz` then the line `a = foo.x;` would not compile because `x` is private in `xyz`.

6.3 ITERATORS

- Since we want to create generic template data structures, we come across the issue of searching through template lists.

- We want a search mechanism that is generic in the same way as the linked list. The method for doing this is to create a template object called an *iterator* and include an instance of this in the declaration of the linked list.
- The iterator has functions for taking us to the beginning and end of the list, plus overloads the increment and decrement operators to allow us to move back and forth along the list.

6.4 dlist.h

```
sally% cat dnode.h
```

```
#ifndef DNODE_H
#define DNODE_H

/*****\
    template node class for doubly linked list
\*****/

template <typename dataType> struct dnode
{
    dataType data;
    dnode *prev, *next;

    // constructors
    dnode() : prev(NULL), next(NULL) {
    }

    dnode(const dataType& dataItem,
          dnode *prevPtr, dnode *nextPtr) :
        data(dataItem), prev(prevPtr),
        next(nextPtr) {
    }
};

#endif
```

```

// #####

sally% cat dlistIterator.h
#ifndef DLISTITERATOR_H_
#define DLISTITERATOR_H_

#include "dnode.h"

/*****\
    template iterator class for doubly linked
    list class
\*****/

template <typename dataType> class dlistIterator
{
    friend class dlist<dataType>;

private:
    dlist<dataType> *parent;
    dnode<dataType> *current;

public:

    // constructor
    dlistIterator(dlist<dataType> *myParent,
        dnode<dataType> *position) :
        parent(myParent), current(position) {
    }

    // overloaded dereference operator
    dataType& operator * () const {
        if (current == NULL) {
            throw std::invalid_argument(
                "Attempting to dereference NULL
                in dlistIterator");
        }
        return current->data;
    }
}

```

```

// overloaded arrow operator
dataType* operator -> () const {
    if (current == NULL) {
        throw std::invalid_argument(
            "Attempting to dereference NULL
            in dlistIterator");
    }
    return &(current->data);
}

// overloaded prefix increment operator
dlistIterator<dataType> operator ++ () {
    if (current == NULL) {
        throw std::invalid_argument(
            "Attempting to advance past end
            in dlistIterator");
    }
    current = current->next;
    return *this;
}

// overloaded postfix increment operator
dlistIterator<dataType> operator ++ (int) {
    dlistIterator<dataType>
        current_data = *this;
    ++(*this);
    return current_data;
}

// overloaded equality operator
bool operator ==
    (const dlistIterator &other) {
    return current == other.current;
}

// overloaded inequality operator
bool operator !=
    (const dlistIterator &other) {
    return current != other.current;
}
};

#endif

```

```

// #####

sally% cat dlist.h

#ifndef DLIST_H_
#define DLIST_H_

#include <stdexcept>

template <typename dataType> class dlist;

#include "dnode.h"
#include "dlistIterator.h"
#include "dlistConstIter.h"

/*****\
    template class for doubly linked list
\*****/

template <typename dataType> class dlist
{
    private:
        dnode<dataType> *head, *tail;
        int numItems;

    public:
        /*****\
            iterator friendship and functions
        \*****/

        friend class dlistIterator<dataType>;
        friend class dlistConstIter<dataType>;

        dlistIterator<dataType> begin() {
            return
                dlistIterator<dataType>(this, head);
        }

        dlistConstIter<dataType> begin() const {
            return
                dlistConstIter<dataType> (this, head);
        }
}

```



```

dlistIterator<dataType> end() {
    return
        dlistIterator<dataType>(this, NULL);
}

/*****\
    constructor and destructor functions
\*****/

// default constructor
dlist() :
    head(NULL), tail(NULL), numItems(0) {
}

// copy constructor
dlist(const dlist<dataType> &other) :
    head(NULL), tail(NULL), numItems(0) {
    for (dlistConstIter<dataType>
        itr = other.begin();
        itr != other.end(); itr++) {
        push_back(*itr);
    }
}

// destructor
~dlist() {
    while(head != NULL) {
        // delete every dnode in list
        dnode<dataType> *current = head;
        head = head->next;
        delete current;
    }
}

/*****\
    misc functions
\*****/

void swap(dlist<dataType> &other) {
    std::swap(numItems, other.numItems);
    std::swap(head, other.head);
    std::swap(tail, other.tail);
}

```

```

bool empty() {
    return (numItems == 0);
}

int size() {
    return numItems;
}

bool find(const dataType &findData,
          dlistIterator<dataType> &itr)
{
    // function for finding something
    // in a dlist

    for (itr = begin();
         itr != end(); itr++) {
        if (*itr == findData) return true;
    }
    return false;
}

/*****\
    push and insertion functions
\*****/

void push_front(const dataType &item) {
    head =
        new dnode<dataType>(item, NULL, head);
    if (!empty()) {
        head->next->prev = head;
    }
    else
    {
        tail = head;
    }
    numItems++;
}

```

```

void push_back(const dataType &item) {
    if (!empty()) {
        tail->next = new dnode<dataType>
                        (item, tail, NULL);
        tail = tail->next;
        numItems++;
    } else {
        push_front(item);
    }
}

dlistIterator<dataType>
insert(dlistIterator<dataType> pos,
const dataType &item) {
    if (pos.current == head) {
        push_front(item);
        return begin();
    } else if (pos.current == NULL) {
        push_back(item);
        return dlistIterator<dataType>
                (this, tail);
    } else {
        dnode<dataType> *newNode =
            new dnode<dataType>
                (item, pos.corrent->prev,
                pos.current);
        pos.current->prev->next = newNode;
        pos.Current->prev = newNode;
        numItems++;
        return dlistIterator<dataType>
                (this, newNode);
    }
}

/*****\
    pop and erase functions
\*****/

```

```

void pop_front() {
    if (empty()) {
        throw std::invalid_argument
            ("Attempting to pop front of
             empty list");
    }

    // slice out the front node
    dnode<dataType> *removeNode = head;
    head = head->next;
    if (head != NULL) {
        head->prev = NULL;
    } else {
        tail = NULL;
    }

    delete removeNode;
    numItems--;
}

void pop_back() {
    if (empty()) {
        throw std::invalid_argument
            ("Attempting to pop back
             of empty list");
    }

    // slice out the back node
    dnode<dataType> *removeNode = tail;
    tail = tail->prev;
    if (tail != NULL) {
        tail->next = NULL;
    } else {
        head = NULL;
    }

    delete removeNode;
    numItems--;
}

```

```

dlistIterator<dataType>
    erase(dlistIterator<dataType> &pos) {
    if (empty()) {
        throw std::invalid_argument
            ("Attempting to erase from
             empty list");
    }
    if (pos == end()) {
        throw std::invalid_argument
            ("Attempting to erase end()");
    }

    dlistIterator<dataType>
        returnIter = pos;
    returnIter++;

    if (pos.current == head) {
        pop_front();
    } else if (pos.current == tail) {
        pop_back();
    } else {
        dnode<dataType>
            *removeNode = pos.current;
        removeNode->prev->next =
            removeNode->next;
        removeNode->next->prev =
            removeNode->prev;
        numItems--;
        delete removeNode;
    }
    return returnIter;
}

/*****
overloaded operators
*****/

// assignment operator
dlist<dataType>& operator =
    (const dlist<dataType> &other) {
    dlist<dataType> tempCopy(other);
    swap(tempCopy);
    return *this;
}

```

```

};

#endif

// #####

sally% cat testmain.cpp

/*****\
    Test program for demonstrating container
    types
*****/

#include <sys/time.h>
#include <time.h>
#include <stdlib.h>
#include <iostream>
#include <algorithm>

#include "dataobject.h"
#include "dlist.h"

using namespace std;

// function prototypes
double difUtime(struct timeval *first,
                struct timeval *second);
template <typename dataType>
    dlistIterator<dataType> findData(
        dlist<dataType> &testContainer,
        const dataType &target);

double difUtime(struct timeval *first,
                struct timeval *second)
{
    // return the difference in seconds,
    // including milli seconds

    double difsec =
        second->tv_sec - first->tv_sec;

    double difmilli =

```

```

        second->tv_usec - first->tv_usec;

    return (difsec + (difmilli) / 1000000.0);
}

template<typename dataType>
dlistIterator<dataType> findData
    (dlist<dataType> &testContainer,
     const dataType &target)
{
    /*****\
        template function for finding something
        in a dlist. return an iterator positioned
        at correct place in dlist or return
        iterator positioned at end of list
    *****/

    for (dlistIterator<dataType>
          itr = testContainer.begin();
          itr != testContainer.end(); itr++) {
        if (*itr == target) return itr;
    }
    return testContainer.end();
}

int main()
{
    const int MAXDATA = 1000000;
    dataObject *doPtr;
    int i, keyvals[MAXDATA];
    dlist<dataObject> testContainer;

    // data for calculating timing
    struct timeval first, second;
    double usecs;

    try {
        /*****\
            Initialise things to demonstrate the
            container
            - fill keyvals and scramble it
        *****/

```

```

for (i=0; i<MAXDATA; i++) keyvals[i] = i;
srand(time(NULL));
for (i=0; i<MAXDATA; i++)
    swap(keyvals[i],
        keyvals[random() % MAXDATA]);
int middle = keyvals[MAXDATA/2];

/*****\
    test inserting MAXDATA data pieces into
    the container with keyval 0 to
    MAXDATA-1 in random order
*****/

gettimeofday(&first, NULL);
for (i=0; i<MAXDATA; i++) {
    doPtr = new dataObject(keyvals[i]);
    testContainer.push_back(*doPtr);
}
gettimeofday(&second, NULL);
usecs = difUtime(&first, &second);
cout << MAXDATA <<
    " items in container in random order\n";
cout << "time taken to push data into
    container = " << usecs << " seconds\n\n";

/*****\
    test finding data in the container
*****/

gettimeofday(&first, NULL);
findData(testContainer, dataObject(0));
gettimeofday(&second, NULL);
usecs = difUtime(&first, &second);
cout << "time taken to find first item in
    container = " << usecs << " seconds\n";

gettimeofday(&first, NULL);
dlistIterator<dataObject> itr =
    findData(testContainer,
        dataObject(keyvals[middle]));
gettimeofday(&second, NULL);
usecs = difUtime(&first, &second);
cout << "time taken to find item in middle

```



```

        of container = " << usecs << " seconds\n";

gettimeofday(&first, NULL);
findData(testContainer,
        dataObject(MAXDATA));
gettimeofday(&second, NULL);
usecs = difUtime(&first, &second);
cout << "time taken to find item doesn't
        exit in container = " << usecs <<
        " seconds\n\n";

/*****\
    test removing data from the container
*****/

gettimeofday(&first, NULL);
testContainer.pop_front();
gettimeofday(&second, NULL);
usecs = difUtime(&first, &second);
cout << "time taken to erase first item in
        container = " << usecs << " seconds\n";

gettimeofday(&first, NULL);
testContainer.erase(itr);
gettimeofday(&second, NULL);
usecs = difUtime(&first, &second);
cout << "time taken to erase middle item in
        container = " << usecs << " seconds\n";

gettimeofday(&first, NULL);
testContainer.pop_back();
gettimeofday(&second, NULL);
usecs = difUtime(&first, &second);
cout << "time taken to erase last item in
        container = " << usecs << " seconds\n";
} catch (out_of_range &ex) {
    cout << "\nERROR - Out of Range Exception
            thrown\n" << ex.what() << "\n";
    exit(1);
} catch(...) {
    cout << "\nERROR - undefined Exception
            thrown\n";
    exit(1);
}

```

```

    return 0;
}

// #####

sally% cat makefile
CC = g++
prog: testmain.o
    $(CC) testmain.o -Wall -o testmain
testmain.o: testmain.cpp dlist.h dataobject.h
    $(CC) -Wall -c testmain.cpp

sally% testmain
1000000 items in container in random order
time taken to push data into container = 2.99964
seconds

time taken to find first item in container =
0.067085 seconds
time taken to find item in middle of container =
0.603338 seconds
time taken to find item doesn't exist in container
= 1.23745 seconds

time taken to erase first item in container =
9.1e-05 seconds
time taken to erase middle item in container =
8e-06 seconds
time taken to erase last item in container =
4e-06 seconds

```