1. Description of the implementation of the linklist :

When each sequence is created, it has a head pointer

that points to nullptr. When a value is being inserted

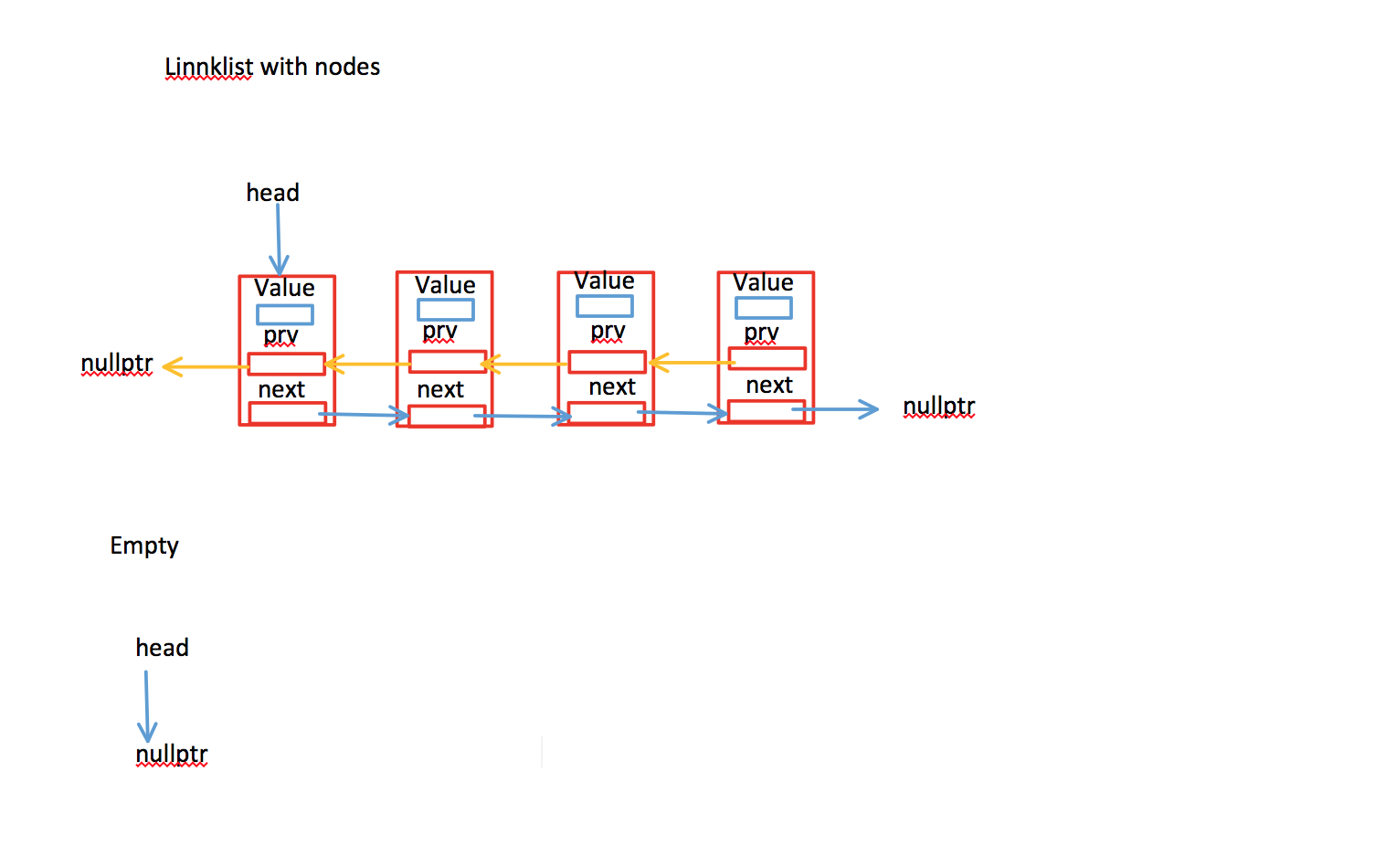
the head pointer will then point to corresponding first

node depend on the call of insert( pos, value ) or insert( value).

In the linklist, the node struct has 3 member variable, they are

Itemtype value, two Node type pointer next and previous.

The previous pointer will point to nullptr for the first node and the next pointer will point to nullptr for the last node.



1. Pseudo code for the implementation of non-trivial functions

Constructor

Sequence::Sequence()

Set the head pointer to nullptr

Set the size of list to 0

Copy Constructor

Sequence::Sequence( const Sequence& other)

Set the head pointer to nullptr

Set the size to 0

Create a temp pointer to travers through the ***other*** linklist

While the temp pointer is not a nullptr

Copy the value for each position in ***other***

And inserted the value at that position in our linklist

Increment the position and update the temp pointer to next

Assignment operator

Sequence& Sequence::operator=( const Sequence& other)

First check if the linklist is copying itself

If not then

Construct a temp sequence with copy-constructor with ***other***

Swap with the temp sequence just created

Return a reference of this

Destructor

Sequence::~Sequence()

First check is the linklist is empty

If not then:

Create a temp pointer to travers down the linklist

While the pointer is not a nullptr

Create another pointer temp2 that keep track of the next node

Delet the current node

Decrement the size

Update the temp pointer with temp2

Insert

int Sequence::insert(int pos, const ItemType& value)

first check if the position is valid

if so then

allocate a new node

store the value

if the linklist is empty

set head pointer to this new node

if the pos is 0

add to front ()

{

let the newnode’s next pointer point to head

update the pervious pointer

set new head

}

if the pos is the size

add to the rare()

{

travers down to the last node

update the next and previous pointer

}

else

add to the middle()

{

traverse down with a temp node

to the corresponding pos while the pointer

is not nullptr

check if the pointer is nullptr

if not then access the pointer

update and reconnect the next and previous pointer

}

update the size

return the pos

int Sequence::insert(const ItemType& value)

{

create a temp pointer to head

travers the pointer

if the value of the visited node is greater and equal to the value pass in

break

update pos and temp pointer

insert the value at that position with the insert function

}

bool Sequence::erase(int pos)

check if the position is valid

create a temp pointer to the head

if the linklist is empty

return false

if the pos is 0

use a pointer to remember the old head

update the head

delete the old head

else

travers the linklist with the pointer to the corresponding position

use a pointer to remember the ***wanted delete node***

disconnect the next and pervious pointer of its pervious and next node

update the next and pervious pointers

delete the ***wanted delete node***

decrement the size

return true

int Sequence::remove(const ItemType& value)

{

while the find function still finds the value

erase the positon returned by find

increment count

return count

}

bool Sequence::get(int pos, ItemType& value) const

{

check if the positon is valid

if so then

travers down the linklist with temp pointer

to the correct location

check if the pointer is nullpter

if not access it and copy in to ***value***

return true

}

bool Sequence::set(int pos, const ItemType& value)

{

check if the positon is valid

if so then

travers down the linklist with temp pointer

to the correct location

check if the pointer is nullpter

if not access it and update the old value to new value

return true

}

int Sequence::find(const ItemType& value) const

{

for the size of a linklist travers down with a pointer

if the pointer is not nullptr and the value of the node is a target

return that position

update the temp pointer

return -1

}

void Sequence::swap(Sequence& other)

{

use a temp pointer to hold the other’s head pointer

set other’s head pointer to this->head

set this head to the temp pointer

use a temp int variable to hold the other linklist’s size

set other’s size to this size()

and set this->size to the temp int variable

}

int subsequence(const Sequence& seq1, const Sequence& seq2)

{

if seq2 is not empty and the size is less then seq1 then

get the first value of seq2

find the first positon inseq1 that the value appears

if the none appears

return -1

else

start from the location in seq1

create a default bool var and set it to true

travers down the size of seq1 with for loop

for the size of seq2

if the element in get inseq1 missmatch

with the element of seq2

set the bool var to false

return the pos if bool var is still true

return -1

void interleave(const Sequence& seq1, const Sequence& seq2, Sequence& result)

get the size of each seq1 and seq2

clear result if it is not empty

compare the sizes and set the length to be larger one

create a int counter ***start*** and set it 0

for the length compute above in each iteration

if we can get the value at i in seq1

insert it to result at ***start***

update counter ***start***

if we can get the value at i in seq2

insert it to result at ***start***

update counter ***start***

1. Test case:a list of test cases that would thoroughly test the functions. Be sure to indicate the purpose of the tests. For example, here's the beginning of a presentation in the form of code:

The tests were performed on a sequence of strings (i.e., ItemType was a type alias for std::string).

// default constructor

Sequence s;

// For an empty sequence:

assert(s.size() == 0); // test size

assert(s.empty()); // test empty

assert(s.remove("paratha") == 0); // nothing to remove

Even if you do not correctly implement all the functions, you must still list test cases that would test them. Don't lose points by thinking "Well, I didn't implement this function, so I won't bother saying how I would have tested it if I had implemented it."

Sequence s;

s.insert(0, "a");

s.insert(1, "b");

s.insert(2, "c");

s.insert(3, "b");

s.insert(4, "e");

s.insert(5, "f");

s.insert(6, "g");

assert( s.insert(s.size(),"LAST") == s.size()-1);

s.insert(1, "b1");

s.insert(2, "c2");

s.insert(5, "c2");

s.insert(0, "FIRST");

// insertion test++++++

assert( s.insert(-1, "BAD") == -1); // bad position test

assert( s.insert(s.size()+1, "BAD") == -1); // bad position test

//if 0 <= pos <= size() in bound insertion test

assert( s.size() == 12);

cout<<"rm g "<<s.remove("g")<<endl;

cout<<"rm f "<<s.remove("f")<<endl;

cout<<"rm b "<<s.remove("b")<<endl;

// remove all g ,f ,b

assert( s.size() == 8); // this test the remove function

s.remove("c2");

assert( s.size() == 12-6);

assert( s.insert("z") == s.size()-1);

// these test the insert( value ) and check if return position is correct

assert(s.insert("a") == 1);

assert(s.insert( "b") == 3);

assert(s.insert( "d")== 6);

assert(s.insert("y") == s.size()-2);

s.set(0, "a"); // these is a preparation for test the set function

s.set(1, "b");

s.set(2, "c");

s.set(3, "b");

s.set(4, "e");

s.set(5, "f");

s.set(6, "g");

s.set(7, "h");

assert(s.set(s.size(), "iiii") == false); // these test the set function for bad input

assert(s.set(-1, "iiii") == false);

assert(s.set(s.size()-3, "x") == true);

cout<< s.size()<<endl;

s.erase(0);

assert( s.find("a") == -1);

s.erase(1);

assert(s.find("c") == -1);

s.dump();

assert(s.find("h") == 5);

assert(s.find("g") == 4);

assert(s.remove("b") == 2);

s.set(0, "a");

s.set(1, "b");

s.set(2, "c");

s.set(3, "b");

s.insert(4, "e");

s.insert(5, "f");

s.insert(6, "g");

s.insert(7, "g");

s.dump();

// these test assertion tested the get function

// and check if the set function is working properly from above setting

ItemType value = "xxx";

assert(!s.get(s.size(), value) && value=="xxx");

assert(s.get(0, value) && value=="a");

assert(s.get(1, value) && value=="b");

assert(s.get(2, value) && value=="c");

assert(s.get(3, value) && value=="b");

assert(s.get(4, value) && value=="e");

assert(s.get(5, value) && value=="f");

assert(s.get(6, value) && value=="g");

assert(s.get(7, value) && value=="g");

assert(s.get(8, value) && value=="x");

assert(s.get(9, value) && value=="y");

assert(s.get(10, value) && value=="z");

assert(!s.get(11, value) && value=="z");

while(!s.empty()) // these tested the empty function and erase function

{

s.erase(0);

cout<<"curr size "<<s.size()<<endl;

}

assert(s.empty());

// same size swap test settings

s.insert(0, "a");

s.insert(1, "b");

s.insert(2, "c");

s.insert(3, "d");

s.insert(4, "e");

s.insert(5, "f");

s.insert(6, "h");

s.insert(6, "g");

Sequence a;

a.insert("1");

a.insert("2");

a.insert("3");

a.insert("4");

a.insert("5");

a.insert("6");

a.insert("7");

a.insert("8");

// "+++++same size swap test+++++"

cout<< "before swap s is "<<endl;

s.dump();

cout<< "before swap a is "<<endl;

a.dump();

s.swap(a);

cout<< "After swap s is "<<endl;

s.dump();

cout<< "After swap a is "<<endl;

dump();

// these assertion makes sure that the swapping is working properly

assert( a.find("a") == 0);

assert( a.find("b") == 1);

assert( a.find("c") == 2);

assert( a.find("d") == 3);

assert( a.find("e") == 4);

assert( a.find("f") == 5);

assert( a.find("g") == 6);

assert( a.find("h") == 7);

assert(s.find("1") == 0);

assert(s.find("2") == 1);

assert(s.find("3") == 2);

assert(s.find("4") == 3);

assert(s.find("5") == 4);

assert(s.find("6") == 5);

assert(s.find("7") == 6);

assert(s.find("8") == 7);

// different size swapping settings

a.remove("a"); // makes the a sequences smaller

a.remove("b");

a.remove("c");

int abeforsize= a.size();

int sbeforsize= s.size();

cout<< "before swap s with size "<<s.size() <<endl;

s.dump();

cout<< "before swap a with size "<<a.size() <<endl;

a.dump();

s.swap(a);

cout<< "after swap s with size "<<s.size() <<endl;

s.dump();

cout<< "after swap a with size "<<a.size() <<endl;

a.dump();

assert(s.size()==abeforsize); ) // this check is different size swapping works by comparing to old size

assert(a.size()==sbeforsize); // this check is different size swapping works by comparing to old size

// settings for 0 size sawpping

while(!s.empty()) // another test for the empty function and erase function

{

s.erase(0);

cout<<"curr size "<<s.size()<<endl;

}

cout<< "before swap s with size "<<s.size() <<endl;

s.dump();

cout<< "before swap a with size "<<a.size() <<endl;

a.dump();

s.swap(a);

cout<< "after swap s with size "<<s.size() <<endl;

s.dump();

cout<< "after swap a with size "<<a.size() <<endl;

1. dump();

// this check is different size swapping works by comparing to old size

assert(s.size()==8);

// this check is different size swapping works by comparing to old size

assert(a.size()==0);

+++++ copy constructor test +++++

Sequence A;

A.insert(0, "jonathan");

assert(A.insert(2, "jonathan") == -1);

A.insert(1, "kevin");

A.insert("jammy");

A.insert( "katherine");

A.insert( "andy");

A.insert( "ban");

A.insert("family");

A.dump();

assert(A.size() == 7); // double check with the insert function

// construct B with A B(A)

Sequence B(A);

B.dump();

cout<<"bs "<<B.size()<<endl;

assert(B.size() == A.size());

// "construct C with A C=A”

Sequence C= A;

C.dump();

assert(C.size() == A.size());

// setting for assignment operator test

assert(C.remove("first") == 0) ;

assert(C.remove("jonathan") == 1) ;

assert(C.remove("andy") == 1) ;

while( !C.empty())

{

C.erase(0);

}

assert( C.empty());

string valueT="xxxxx";

assert(!C.get(0, valueT));

assert( valueT == "xxxxx");

C.insert("11");

C.insert("11");

C.insert("12");

C.insert("13");

C.insert("14");

assert(C.insert("11") == 0);

C.dump();

assert( C.insert("14")== C.size()-2);

assert(C.insert("15") == C.size()-1);

assert(C.insert("15") == C.size()-2);

assert(C.get(0, valueT) && valueT =="11");

assert(C.get(C.size()-1, valueT) && valueT =="15");

// visualization of the test

cout<<"Before Assingment opt c is : ";

C.dump();

cout<<"Before Assingment opt a is : ";

A.dump();

A=C;

cout<<"After Assingment opt c is : ";

C.dump();

cout<<"After Assingment opt a is : ";

1. dump();

<<================= some sawp test===============>>

cout<< "Before Swap A and B, A is ";

A.dump();

cout<< "Before Swap A and B, B is ";

B.dump();

B.swap(A);

cout<< "After Swap A and B, A is ";

A.dump();

cout<< "After Swap A and B, B is ";

B.dump();

// this test is a series of consecutives result from above

// since C is being assigned to A

// A is identical to C

// then I swap B with A, which means B now should be identical to C

// thus test below

string vB="BBB", vC="CCC";

assert( B.find("11") == C.find("11"));

assert( B.find("13") == C.find("13"));

assert( B.find("15") == C.find("15"));

for( int i =0 ; i < C.size() ; i++) // this make sure C and B is identical with the values

assert( B.get(i, vB) && C.get(i, vC) && vB==vC);

assert( !B.get(B.size(), vB)); // some test for get and set

assert( !B.get(-1, vB) );

cout<<" ++++++ Set Test +++++\n";

cout<<B.size()<<endl;

B.dump();

assert(B.set( 0, "AB"));

assert(B.set( 1, "CD"));

assert(B.set( 2, "EF"));

assert(B.set( 4, "GH"));

assert(B.set( 5, "GH"));

assert(B.set( 6, "GH"));

assert(B.set( 7, "ZX"));

assert(!B.set( B.size(), "XXXXX"));// bad inputs for set test

assert(!B.set( -1, "XXXXXX")); // bad inputs for set test

assert( B.find("11") != C.find("11")); // now b is different then c because it is modified

assert( B.find("13") != C.find("13"));

assert( B.find("15") != C.find("15"));

assert(B.remove("GH") == 3); // remove test

assert(B.size() == 9-3); // remove test

B.dump();

cout<<" Assign B to A "<<endl;

cout<<" A was : ";

A.dump();

cout<<" B was : ";

B.dump();

A=B;

cout<<" A is : ";

A.dump();

cout<<" B is : ";

B.dump();

for( int i =0 ; i < B.size() ; i++)

assert( B.get(i, vB) && A.get(i, vC) && vB==vC);

assert(!B.erase(C.size()));

assert(!B.erase(-1));

assert(B.remove("11") == 0);

assert(B.remove("12") == 1);

assert(A.remove("11") == 0);

assert(A.remove("12") == 1);

cout<< "After removing 11 and 12 "<<endl;

A.dump();

B.dump();

//"+++++++++++ subsequence test +++++++++++”

assert(subsequence(A, B) == 0);

// since A and B are identical from the previous assignment operator, thus should return 0

assert(subsequence(C, B) == -1); // no common element test

assert(subsequence(A, C) == -1); // no common element test

Sequence s1;

s1.insert(0, "30");

s1.insert(1, "21");

s1.insert(2, "63");

s1.insert(3, "42");

s1.insert(4, "17");

s1.insert(5, "63");

s1.insert(6, "17");

s1.insert(7, "29");

s1.insert(8, "8");

s1.insert(9, "32");

Sequence s2;

s2.insert(0, "63");

s2.insert(1, "17");

s2.insert(2, "29");

assert(subsequence(s1, s2) == 5); // common element test

s2.set(0, "17");

s2.set(1, "63");

s2.set(2, "29");

assert(subsequence(s1, s2) == -1); // no common element test

while( !s2.empty())

s2.erase(0);

assert(subsequence(s1, s2) == -1); // case where seq2 is empty and thus return -1

Sequence c,d;

c.insert(0, "1");

c.insert(1, "2");

c.insert(2, "3");

c.insert(3, "2");

c.insert(4, "1");

c.insert(5, "2");

c.insert(6, "3");

c.insert(7, "4");

c.insert(8, "1");

c.insert(9, "2");

c.insert(10, "3");

c.insert(11, "4");

d.insert(0, "1");

d.insert(1, "2");

d.insert(2, "3");

d.insert(3, "4");

// this tested if the subsequence return the earliest common sequence found

assert(subsequence(c, d) == 4);

// no common element test

assert(subsequence(c, s2) == -1);

<<"+++++++++++ same size interleave test +++++++++++">>

s2.insert(0, "30a");

s2.insert(1, "21a");

s2.insert(2, "63a");

s2.insert(3, "42a");

s2.insert(4, "17a");

s2.insert(5, "63a");

s2.insert(6, "17a");

s2.insert(7, "29a");

s2.insert(8, "8a");

s2.insert(9, "32a");

Sequence res;

res.insert(0, "a");

res.insert(1, "b");

res.insert(2, "c");

interleave(s1, s2, res);

int count = 0;

s1.dump(); // visualization

s2.dump(); // visualization

res.dump();// visualization of the result

// this checked if result is corrected for requirement

for( int i =0 ; i < (s1.size()+s2.size()) ; i+=2) {

ItemType v1,v2, v3;

s1.get(count, v1) ;

s2.get(count, v2);

count++;

assert( res.get(i, v3) && v3==v1 );

assert( res.get(i+1, v3) && v3==v2 );

}

<<"+++++++++++ diff size interleave test +++++++++++">>

s2.erase(6);

s2.erase(5);

s2.erase(4);

s2.erase(3);

interleave(s1, s2, res);

assert(res.size() == ( s1.size() + s2.size())); // this check if interleave works properly

s1.dump();

s2.dump();

res.dump();// visualization of the result

<<"+++++++++++ s1 empty size interleave test +++++++++++">>

Sequence empty1;

cout<< "before interleave res is "<<endl;

res.dump();

interleave(empty1, s2, res);

assert(res.size() == s2.size());

for( int i =0 ; i < s2.size(); i++)

{

ItemType v2, v3;

s2.get(i, v2);

// res should be identical to seq2 since seq1 is empty

assert( res.get(i, v3) && v3==v2 );

}

cout<< "\nafter interleave res is "<<endl;

res.dump();

<<+++++++++++ s2 empty size interleave test +++++++++++">>

cout<< "\nbefore interleave res is "<<endl;

res.dump();

interleave(s1, empty1, res);

assert(res.size() == s1.size());

for( int i =0 ; i < s1.size(); i++)

{

ItemType v1, v3;

s1.get(i, v1);

// res should be identical to seq1 since seq2 is empty

assert( res.get(i, v3) && v3==v1 );

}

cout<< "\nafter interleave res is "<<endl;

res.dump();

<<"+++++++++++ s1 and s2 empty interleave test +++++++++++">>

Sequence empty2;

cout<< "\nbefore interleave res is "<<endl;

res.dump();

interleave(empty1, empty2, res);

// res should be empty even if the pass in res is not since seq1 and seq2 are empty

assert(res.size() == 0 && res.empty());