Name: Linh Tang

CS32

Prof. David A. Smallberg

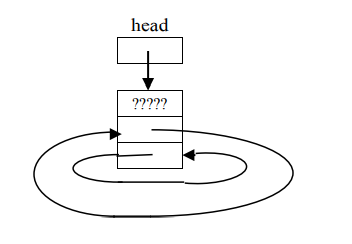
Date:01/26/2021

**Project 2**

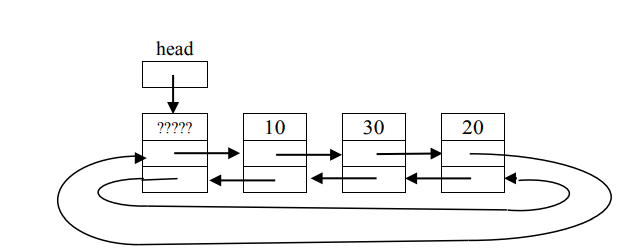
**Circular doubly linked list with dummy node**

In this project, the list is built using a circular doubly linked list with a dummy node. The dummy node is the first node but is not counted as a list node. The first node in the list is the node right after the dummy node. Each node includes three parts: an ItemType value, a pointer points to the next node and a pointer points to the previous node. The previous node of the dummy node is the last node in the last; the next node of the last node is the dummy node.

Empty sequence (there’s only dummy node): (from [Supplement to the linked list lecture](http://web.cs.ucla.edu/classes/winter21/cs32/l2/SupplementLinkedLists.pdf))



A typical sequence



**Pseudocodes:**

**Constructor: Sequence::Sequence()**

The constructor creates an empty sequence containing a dummy node.

* Initialize the size for the empty sequence which is 0.
* Pointing head pointers to the address of the dummy node.
* Setting the next pointer and previous pointer points to the address of the dummy node (where the head pointer points to).

**Copy constructor: Sequence(const Sequence& other)**

* Using a for loop to get data from other sequence and insert them into the current sequence.
* Setting the size of the current sequence equal to the size of other sequence.

**Assignment operator: Sequence::operator=(const Sequence& rhs)**

* If this sequence is different with the rhs sequence, we do:
* Copying the rhs to a new temp sequence by using the copy constructor.
* Swapping this sequence with the temp sequence.

**Destructor: ~Sequence()**

* If the sequence is empty, do nothing.
* If the sequence is not empty, using two pointers: one points to the node we want to delete (first node), the other node is the Next node (deleting from the beginning of the sequence).
* Pointing the head pointer to the Next node; the previous node of the Next node is the head, not the first node.
* Deleting the first node.
* The Next node now becomes the first node in the sequence.
* Repeat this until the last node.

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

This function inserts value into a node at a given position.

* If the pos (less than 0 or greater than the size) is invalid, return -1.
* Creating a new node and pasting the value into it.
* If the sequence is empty, setting up the pointers such as the new node is the node after the dummy node. The next and previous pointer of the dummy node points to the new node; the next and previous pointer of the new node points to the dummy node.
* When the sequence is not empty and we want to insert at the beginning (pos ==0), we reset the pointers so that the new node becomes the first node which is right after the dummy node.
* When the sequence is not empty and we want to insert at the end (pos == size()),the pointers are arranged in a correct way such that the new node becomes the last now.
* When the insertion is occured in the middle of the list, we need two pointers next to each other(for previous and next node)and move together until the second reaches the position we want to insert, arranging pointers in a way that the new node is in between the previous node and the next node.
* After inserting, the size is incremented and then the function returns the pos where the value is inserted into.

**int Sequence::insert(const ItemType& value)**

* Finding the position of the node where the data is greater than or equal to the value. A new node with the value will be inserted into this position, then return the position.
* If we cannot find the position which satisfies the condition, returns -1.

**bool Sequence::erase(int pos)**

* When the pos is invalid or the sequence is empty, returns false.
* Getting the pointer points to the corresponding position.
* If the node we want to erase is the last node, creating a temp node for the previous node of the last node. Rearranging pointers of the temp node and the dummy node so that the temp node becomes the new last node of the sequence. Then, we delete the last node.
* If the node is not the last node, creating two pointers point to the previous and the next nodes of the node we want to delete. Rearranging pointers between these two nodes to make them become consecutive nodes. Then, we delete the node we want to delete.

**int Sequence::remove(const ItemType& value)**

* Getting the positions where the value is found in the sequence.
* Initializing a variable to count the number of times that value appears in the sequence.
* When the value is found, we erase it and then increment the counter variable.
* Returning the counter.

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

* If the pos is invalid, do nothing and return false.
* If not, getting the node pointer is corresponding to that position, then changing the data of the node that pointer points to. Returning true.

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

* If the pos is invalid, do nothing and return false.
* If not, getting the node pointer corresponds to that position, then obtaining the data of the node that pointer points to, pastes it to the value parameter. Returning true.

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

* Getting a node pointer going through all the nodes in the sequence, this pointer needs to stop at the node where it finds the data of the node is the same with the value.
* Returning the position of that node if it is found. If not, returning -1.

**void Sequence::swap(Sequence& other)**

* Using a temporary integer variable to swap the sizes of two sequences.
* Using a temporary node pointer to swap the head pointers of two sequences.

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

* If seq2 is empty, returns -1.
* Using a for loop which goes through all the items in the sequence 1.
* Using get function every time we need to obtain data from seq1 or seq2.
* When the loop finds the item in the seq1 which is matched with the first item in the seq2. we will obtain and check next items in the seq 2 with sequence seq1 to see if the seq2 is a consecutive of seq1. If it is, the function returns the earliest position where that happens. If not, returning -1.
* The function also returns -1 if k(the index of seq1) + seq2.size() > seq1.size().

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

* If seq1 is empty, setting result equal to seq2.
* If seq2 is empty, setting result equal to seq1.
* Using a copy constructor to make copies of seq1, and seq2. We work on these two copies for the rest of the function.
* The result needs to be emptied if it is not.
* If the size of seq2 is greater than seq1, we use a for loop based on the size of seq2 to get the data from two sequences. If the value can be obtained by using a get() function, we insert that value into the result.
* If the size of seq2 is less than or equal to seq1, we do the similar thing but the for loop now follows the size of seq1.

**Test cases:**

1. **Copy constructor: Sequence(const Sequence& other) and insert string**

Sequence sa;

sa.insert(0, "a");

sa.insert(1, "b");

sa.insert(2, "c");

sa.dump();

Sequence sb(sa);

sb.dump();

1. **Assignment operator: Sequence::operator=(const Sequence& rhs)**

Sequence sa;

sa.insert(0, 1);

sa.insert(1, 2);

sa.insert(2, 3);

sa.dump();

Sequence sb;

Sb = sa;

sb.dump();

1. **Bool empty() const**

Sequence s;

assert(s.empty() == true);

s.insert(0, 3);

assert(s.empty() == false);

1. **Int size() const**

Sequence s;

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

s.insert(0, 3);

s.insert(1, 5);

s.insert(2, 7);

assert(a.size() == 3);

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

**And int Sequence::insert(const ItemType& value)**

Sequence s1;

for (int i = 0; i < 8; i++)

{

s1.insert(i, i \* 10);

}

s1.dump();

s1.insert(15);

s1.dump();

1. **bool Sequence::erase(int pos) and int Sequence::remove(const ItemType& value)**

**Sequence ss; // ItemType is std::string**

Sequence ss;

ss.insert(0, "aaa");

ss.insert(1, "bbb");

ss.insert(2, "ccc");

ss.insert(2, "ddd");

ss.insert(1, "aaa");

ss.erase(2);

ss.dump();

cout << ss.remove("aaa") << endl;

ss.dump();

1. **bool Sequence::set(int pos, const ItemType& value) and bool Sequence::get(int pos, ItemType& value) const**

Sequence ss;

ss.insert(0, "aaa");

ss.insert(1, "bbb");

ss.set(0, "eee"); // test for set function

ss.dump();

string value;

ss.get(0, value); // test for get function

cout << value << endl;

1. **int Sequence::find(const ItemType& value) const**

Sequence s;

s.insert(0, 20);

s.insert(1, 24);

s.insert(2, 35);

s.insert(3, 40);

s.insert(4, 35);

assert(s.find(35) ==2);

assert(s.find(5) == -1);

assert(s.find(40) == 3);

1. **void Sequence::swap(Sequence& other)**

Sequence ss; // ItemType is std::string

ss.insert(0, "aaa");

ss.insert(1, "bbb");

ss.insert(2, "ccc");

Sequence sa;

sa.insert(0, "a");

sa.insert(1, "b");

sa.insert(2, "c");

cout << "sa: ";

sa.dump();

ss.swap(sa); // test for swap

cout << "ss: ";

ss.dump();

cout << "sa: ";

sa.dump();

1. **int subsequence(const Sequence& seq1, const Sequence& seq2) and void interleave(const Sequence& seq1, const Sequence& seq2, Sequence& result)**

Sequence s1, s2, s3;

for (int i = 0; i < 8; i++)

{

s1.insert(i, i \* 10);

}

cout << "s1: ";

s1.dump(); // 0 10 20 30 40 50 60 70

for (int j = 0; j < 2; j++)

{

s2.insert(j, j\*10+20);

}

cout << "s2: ";// 20 30 40

s2.dump();

for (int k = 0; k < 2; k++)

{

s3.insert(k, k\*10);

}

cout << "s3: ";// 0 10 20

s3.dump();

cout << subsequence(s1, s2) << endl; // test for subsequence function

cout << subsequence(s1, s3) << endl; // test for subsequence function

// Test for interleave function

Sequence result;

interleave(s1, s2, result);

result.dump();

interleave(result, s2, result);// aliasing

result.dump();