assignment 1

Due Date: Feb 24, 2020

Assignment Completion

In order to for this assignment to be considered completed you must submit:

- a test verification for part 1, part 2 and part 3
- your program for part 1, part 2 and part 3 via github

To get full marks, please make sure you follow the steps listed for assignment submission fully

Assignment Objectives:

In this assignment, you will:

- Implement an abstract data type (ADT) called a Disjoint Set (using a linked list implementation... there are more advance ways to implement this ADT but your assignment must use the linked list version)
- Use a disjoint set to create a maze
- Write a **RECURSIVE** function to run through the maze

Restrictions

No part of the Standard Library is allowed to be used for this assignment

Part 1 Disjoint Set (15 marks)

Starter files:

All starter files can be found in your assignment group repos. They are also available from course repo if you need a copy of the originals

- timer.h
- timer.cpp
- disjointset.h
- disjointset.cpp
- a1q1tester.cpp (see comment at top of file for command line compilation)

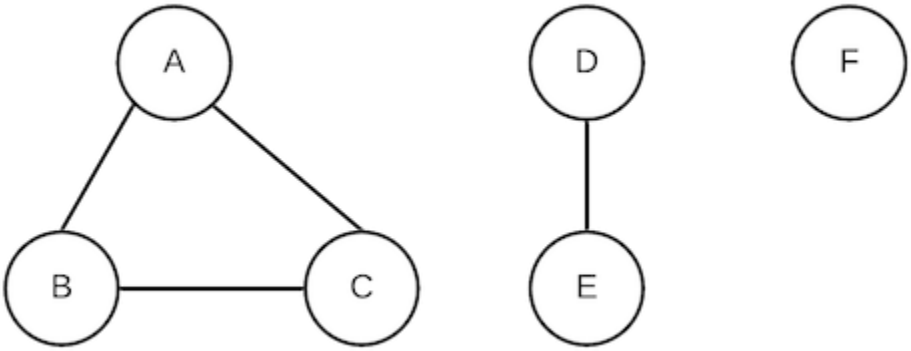
Disjoint Set description

The disjoint set S is a set of non-empty sets = {S_1, S_2, S_3...S_k} where no member of S_i exist in any other set S_j,where i != j. That is, S_i and S_j have no members in common. Each set has an unique identifying member called the representative. A disjoint set ADT has the following operations:

- `makeSet(object);` - creates a new set where ***object*** is the only member of the set. ***object*** is the representative of this new set as it is the only member. each object must be uniquely identifiable (ie no two objects in the disjoint set are the same)
- `findSet(object);` - returns the representative of the set containing ***object***
- `unionSets(object1, object2);` - given two objects, unionSets combines the two sets into one set such that the new set contains every member of two original as long as object1 and object2 were in two separate sets at the start.

The disjoint set in an abstract data type which means that while the above 3 operations describe what it does, it does not speak to any particular implementation. For the purposes of this assignment, we will implement our disjoint set by using linked lists.

Disjoint sets can be useful for various algorithms. For example it can be used to find the connected components within a graph (something we will look at towards the end of the course). Here is a graph with 3 different connected components.



to determine which vertices(the circles) are connected, we start by forming a disjoint set for every vertex:

{{A}, {B}, {C}, {D}, {E}, {F}}.

Then for each of the edges (the lines) we join the associated vertices together if not already joined. The edges are:

- A-B
- B-C

- A-C
- D-E

Basic algorithm

```
for each vertex v{
   //create a separate set for each vertex
   makeSet(v);
}
for each edge e{
   let v1 and v2 represent vertices of edge e;
   rep1=findSet(v1);
   rep2=findSet(v2);
   if(rep1!=rep2){
      unionSets(rep1,rep2);
   }
}
```

edge	rep1	rep2	disjoint set
Initial:			{{A}, {B}, {C}, {D}, {E}, {F}}.
A-B	A	B	{{A,B},{C},{D},{E},{F}}
B-C	A	C	{{A,B,C}, {D},{E},{F}}
A-C	A	A	{{A,B,C}, {D},{E},{F}}
D-E	D	E	{{A,B,C}, {D,E},{F}}

And thus we end up with 3 sets, each set representing the vertices that are connected. In any case, this example is just here to illustrate what a disjoint set is and how it can be used to solve a problem.

The DisjointSet Class

To simplify our implementation of the DisjointSets class, we will identify each object by a unique number starting from 0. There are better ways that you might have to implement a disjoint set for other types of identifiers but for our purposes a number is good enough. In otherwords, we view every member that can be in any disjoint set as single number, our objects will be just a number.

When a DisjointSets is instantiated, it is passed a number representing the maximum number of unique items in the disjoint set. Thus the objects for the disjoint set would be a number ranging from 0 to max-1. No object in the set will have a value outside this range.

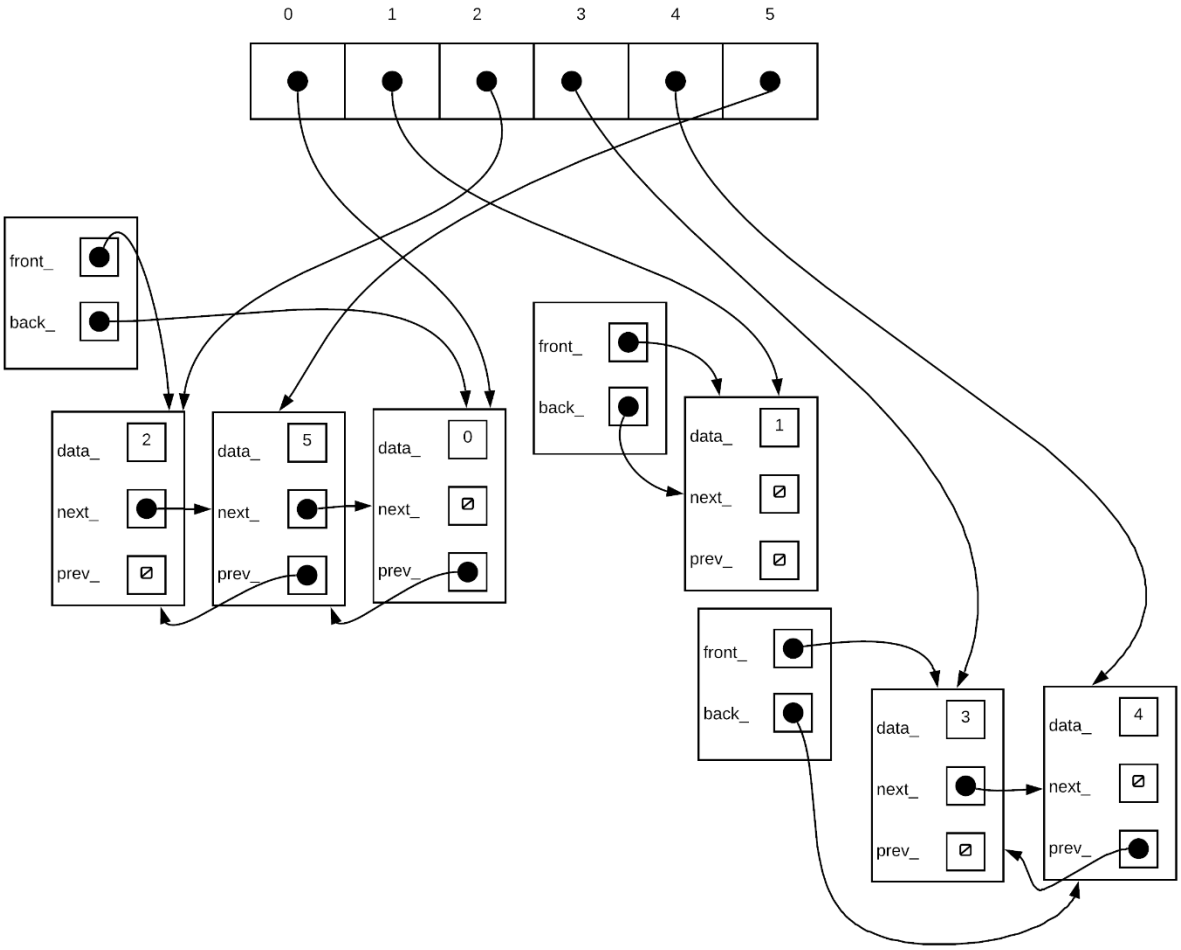
Each of the sets within the disjoint sets are represented by a separate linked list. Each node within the linked lists contain a member of the disjoint set.

Our disjoint set will be made of an array of Node*. When a set is first created using `makeSet()`, we instantiate a node and form a linked list with just that node. The `unionSets()` function thus combines two existing linked list into one.

Example:

Suppose you had the following disjoint set: { {0,2,5},{1},{3,4}}

Our representation would create a structure similar to the following: **Note this diagram is just a very rough guide. You are allowed to change the exact nature of the list. The list can be singly linked or circular for example. You are also allowed to add extra information to each node in order to improve performance. You are even allowed to make the linked list without the "List" header object (ie just nodes linked together, use nullptrs to indicate front/back). You are allowed to add other members to the disjointSet class to help support your processing also. This diagram is only a very rough guide. **



Member functions

```
bool makeSet(int object);
```

- Creates a new set with object as the only member of the set
- if object is out of range (not between 0 and max-1) or object is already part of a set, function does nothing and returns false
- otherwise function creates a set with object as its only member and returns true
    - in other words this will form a linked list with one node who's value is object
    - this one value is also the representative of the set

```
int findSet(int object) const;
```

- returns the representative of the set containing object
- if object is not part of any set or is invalid, function returns -1
- Note that any member can be chosen as the representative. However, once chosen, the representative cannot change unless the membership of the set is changed (for example when unionSets() is performed).

```
bool unionSets(int object1, int object2);
```

- this combines the two sets of object1 and object2
- if either object1 or object2 are not in any set or is invalid, function does nothing and returns false
- if object 1 and object 2 are in the same set, function does not nothing and returns true
- if object 1 and object 2 are in different sets:
    - function combines the two sets (joins together the two linked lists)
    - A new representative is chosen. This new representative can be any member of the new set. That is a call to findSet() using any of the original members would result in the new representative.
    - function returns true

Aside from the above functions, you need to also implement the following:

- copy constructor
- move constructor
- assignment operator
- move assignment operator
- destructor

Efficiency

For this particular ADT, there are a few things you can do to make the data structure more efficient. It is up to you to do some research on this. Part of your mark is determined by the efficiency of your implementation. **NOTE that you must use a linked list representation. A tree/forest representation will not be accepted.**

Part 2: Maze Creator (5 marks)

Starter Files

All starter files can be found in your assignment group repos. They are also available from course repo if you need a copy of the originals

- timer.h
- timer.cpp
- wall.h
- a1q2.cpp
- a1q2tester.cpp (see comment at top of file for command line compilation)

Description

In this part of the assignment, you will use the disjoint set you wrote earlier to create a maze.

Maze creation:

We can create a random maze by making use of a disjoint set. Write the function:

```
int generateMaze(int row, int col, Wall walls[]);
```

- ***row*** and ***col*** describe the size of the maze in terms of the number of cells along its rows and columns.
- function will generate a list of Wall objects that form the maze and pass those back to the calling function using the ***walls*** array.
- function return the number of walls in the final maze

We describe the maze as having row X col cells. For example if row was 3, and col was 4, then we would have a grid of cells as follows. We describe a wall by the two cell numbers the wall separates (smaller number first). If every single wall existed, there would be (row-1)(col) + (col-1)(row) walls.

```
 0 |  1 |  2 |  3
-------------------
 4 |  5 |  6 |  7
---------------------
 8 |  9 | 10 | 11
```

The way maze generation works is as follows:

- begin by generating every possible wall that can exist between the cells (don't generate the outer borders, that actually just makes it harder)
- starting with that list of walls, select a random wall. See if the two cells on either side of the walls are already connected. If not, remove the wall between them (it does not go into the final set of walls). If the two cells are already joined, do not remove the wall.
- continue the above process until every cell is joined.

Remember, that your disjoint set from part 1 could help you solve this problem!

Part 3: Maze Runner (5 marks)

Starter Files

All starter files can be found in your assignment group repos. They are also available from course repo if you need a copy of the originals

- timer.h
- timer.cpp
- wall.h
- maze.h
- maze.cpp
- a1q3.cpp
- a1q3tester.cpp (see comment at top of file for command line compilation)

Description

Write a recursive maze runner:

```
int runMaze(Maze& theMaze, int path[], int fromCell, int toCell);
```

The `runMaze()` function will find a path from cell number ***fromCell*** to cell number ***toCell***. function will "markup" theMaze with the path and pass back an array containing the path (using the cell numbers) from the starting cell to ending cell. The function will return the number of cells along the pathway from the starting cell to the ending cell inclusive.

Assignment submission

Submit a test verification

A1 Part 1 submitter command:

```
~catherine.leung/submit dsa555-A1Part1
```

A1 Part 2 submitter command:

```
~catherine.leung/submit dsa555-A1Part2
```

A1 Part 3 submitter command:

```
~catherine.leung/submit dsa555-A1Part3
```

Submit your code

- push all your code into your team github repository for a1 (just into the master branch, do not use folders as only a1 files will be in the repo).
- Each member of the team must submit their own code under their own account (ie if you wrote it, and we look at the code we should see that it was you who submitted it not your partner...can be done by looking hitting the blame button on a source file). Your teammates should not be pushing code into the repo on your behalf. Learning to use enough git to do this is good for you. If you don't love the command line use github desktop.

Late penalties

- up to 1 week late (and yes, break week does count), 10%
- 10% per day there after to a maximum of 50%

Resubmissions

With the test verification there is very rarely a need to have you resubmit your program. However, there are instances where the work may pass testing but still miss the point of the

assignment entirely. In those cases, your work may need to be resubmitted. Here are some examples:

- doing a recursive function iteratively
- using a totally different data structure than specified (a forest instead of a linked list for disjoint set for example)
- writing a data structure in a manner that clearly demonstrates a lack of understanding of how the data structure works

Any work that is resubmitted, will receive a 50% penalty. NOTE: Any work with a grade of resubmit is considered to be not completed.

Coding Grading Breaking

- Documentation - 20%
- Coding Style - 10%
- Memory Management - 20%
- Efficiency - 25%
- Completion and correctness of functions/functionality - 25% - Some aspects of some functions may not be tested by the tester. Sometimes this is because it simply isn't possible (like destructors for example). Thus, we reserve the right to read your program and take off marks for anything that does not work to spec.