SPA_pico

Generated by Doxygen 1.9.4

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

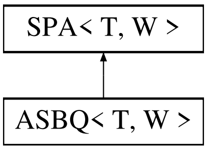
Chapter 4

Class Documentation

4.1 ASBQ< T, W > Class Template Reference

```
#include <ASBQ.h>
```

Inheritance diagram for ASBQ< T, W >:



Public Member Functions

- ASBQ (T maxRow, T maxCol, Maze< T, W > &maze)
- ∼ASBQ ()
- void findSP () override
- W getShortestPathLength () const override
- std::string getTypeName () const override

Additional Inherited Members

4.1.1 Detailed Description

template<typename T, typename W>
class ASBQ< T, W >

A∗ algorithm implementation class that finds the shortest path of given maze using a bucket queue. The data types are essential for optimizing the space complexity.

Template Parameters

T	The data type of row and column of maze.
W	The data type of the maze's weights.

4.1.2 Constructor & Destructor Documentation

4.1.2.1 ASBQ()

```
template<typename T , typename W >
ASBQ< T, W >::ASBQ (
             T maxRow,
             T maxCol,
             Maze< T, W > & maze )
```

Constructor for A∗ algorithm class with bucket queue.

Parameters

maxRow	Max row size of the given maze.
maxCol	Max column size of the given maze.
maze	The reference variable for the maze.

4.1.2.2 ∼ASBQ()

```
template<typename T , typename W >
ASBQ< T, W >::∼ASBQ
```

Destructor for A∗ algorithm class with bucket queue.

4.1.3 Member Function Documentation

4.1.3.1 findSP()

```
template<typename T , typename W >
void ASBQ< T, W >::findSP ( )  [override], [virtual]
```

Find the shortest path from the starting point to the ending point in the maze.

Implements SPA< T, W >.

4.1.3.2 getShortestPathLength()

```
template<typename T , typename W >
W ASBQ< T, W >::getShortestPathLength ( ) const  [inline], [override], [virtual]
```

Getter for the length of the shortest path found.

Returns

The length of the shortest path found.

Implements SPA< T, W >.

4.1.3.3 getTypeName()

```
template<typename T , typename W >
std::string ASBQ< T, W >::getTypeName ( ) const  [inline], [override], [virtual]
```

Getter for the name of the algorithm for finding the shortest path in a maze.

Returns

A name of the current algorithm.

Implements SPA< T, W >.

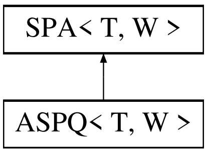
The documentation for this class was generated from the following file:

- SPA/ASBQ.h

4.2 ASPQ< T, W > Class Template Reference

```
#include <ASPQ.h>
```

Inheritance diagram for ASPQ< T, W >:



Public Member Functions

- ASPQ (T maxRow, T maxCol, Maze< T, W > &maze)
- ~ASPQ ()
- void findSP () override
- W getShortestPathLength () const override
- std::string getTypeName () const override

Additional Inherited Members

4.2.1 Detailed Description

template<typename T, typename W>
class ASPQ< T, W >

A∗ algorithm implementation class that finds the shortest path of given maze using a priority queue. The data types are essential for optimizing the space complexity.

Template Parameters

T	The data type of row and column of maze.
W	The data type of the maze's weights.

4.2.2 Constructor & Destructor Documentation

4.2.2.1 ASPQ()

```
template<typename T , typename W >
ASPQ< T, W >::ASPQ (
             T maxRow,
             T maxCol,
             Maze< T, W > & maze )
```

Constructor for A∗ algorithm class with priority queue.

Parameters

maxRow	Max row size of the given maze.
maxCol	Max column size of the given maze.
maze	The reference variable for the maze.

4.2.2.2 ∼ASPQ()

```
template<typename T , typename W >
ASPQ< T, W >::∼ASPQ
```

Destructor for A∗ algorithm class with priority queue.

4.2.3 Member Function Documentation

4.2.3.1 findSP()

```
template<typename T , typename W >
void ASPQ< T, W >::findSP ( )  [override], [virtual]
```

Find the shortest path from the starting point to the ending point in the maze.

Implements SPA< T, W >.

4.2.3.2 getShortestPathLength()

```
template<typename T , typename W >
W ASPQ< T, W >::getShortestPathLength ( ) const  [inline], [override], [virtual]
```

Getter for the length of the shortest path found.

Returns

The length of the shortest path found.

Implements SPA< T, W >.

4.2.3.3 getTypeName()

```
template<typename T , typename W >
std::string ASPQ< T, W >::getTypeName ( ) const  [inline], [override], [virtual]
```

Getter for the name of the algorithm for finding the shortest path in a maze.

Returns

A name of the current algorithm.

Implements SPA< T, W >.

The documentation for this class was generated from the following file:

- SPA/ASPQ.h

4.3 BucketQueue< Key, Value > Class Template Reference

```
#include <BucketQueue.h>
```

Public Member Functions

- BucketQueue ()
- ∼BucketQueue ()
- void push (Key key, Value &value)
- void pop ()
- Value top ()

4.3.1 Detailed Description

template<class Key, class Value>
class BucketQueue< Key, Value >

Bucket queue class that sorts values using a key as an index.

Template Parameters

Key	Data type for key.
Value	Data type for Value.
T	Data type for index.

4.3.2 Constructor & Destructor Documentation

4.3.2.1 BucketQueue()

```
template<class Key , class Value >
BucketQueue< Key, Value >::BucketQueue
```

Constructor for bucket queue.

4.3.2.2 ∼BucketQueue()

```
template<class Key , class Value >
BucketQueue< Key, Value >::∼BucketQueue
```

Destructor for bucket queue.

4.3.3 Member Function Documentation

4.3.3.1 pop()

```
template<class Key , class Value >
void BucketQueue< Key, Value >::pop
```

Delete the data at the top.

4.3.3.2 push()

```
template<class Key , class Value >
void BucketQueue< Key, Value >::push (
             Key key,
             Value & value )
```

Push a new key, value pair to the queue.

Parameters

key	A key of new data.
value	A value of new data.

4.3.3.3 top()

```
template<class Key , class Value >
Value BucketQueue< Key, Value >::top
```

Return the value of data at the top.

Returns

The value of data at the top.

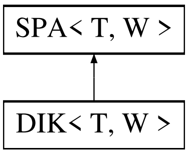
The documentation for this class was generated from the following file:

- structure/BucketQueue.h

4.4 DIK< T, W > Class Template Reference

```
#include <DIK.h>
```

Inheritance diagram for DIK< T, W >:



Public Member Functions

- DIK (T maxRow, T maxCol, Maze< T, W > &maze)
- ~DIK ()
- void findSP () override
- W getShortestPathLength () const override
- std::string getTypeName () const override

Additional Inherited Members

4.4.1 Detailed Description

template<typename T, typename W>
class DIK< T, W >

Naive implementation class of the dijkstra algorithm. The data types are essential for optimizing the space complexity.

Template Parameters

T	The data type of row and column of maze.
W	The data type of the maze's weights.

4.4.2 Constructor & Destructor Documentation

4.4.2.1 DIK()

```
template<typename T , typename W >
DIK< T, W >::DIK (
             T maxRow,
             T maxCol,
             Maze< T, W > & maze )
```

Constructor for Dijkstra algorithm class.

Parameters

maxRow	Max row size of the given maze.
maxCol	Max column size of the given maze.
maze	The reference variable for the maze.

4.4.2.2 ∼DIK()

```
template<typename T , typename W >
DIK< T, W >::∼DIK
```

Destructor for dijkstra algorithm class.

4.4.3 Member Function Documentation

4.4.3.1 findSP()

```
template<typename T , typename W >
void DIK< T, W >::findSP ( )  [override], [virtual]
```

Find the shortest path from the starting point to the ending point in the maze.

Implements SPA< T, W >.

4.4.3.2 getShortestPathLength()

```
template<typename T , typename W >
W DIK< T, W >::getShortestPathLength ( ) const  [inline], [override], [virtual]
```

Getter for the length of the shortest path found.

Returns

The length of the shortest path found.

Implements SPA< T, W >.

4.4.3.3 getTypeName()

```
template<typename T , typename W >
std::string DIK< T, W >::getTypeName ( ) const  [inline], [override], [virtual]
```

Getter for the name of the algorithm for finding the shortest path in a maze.

Returns

A name of the current algorithm.

Implements SPA< T, W >.

The documentation for this class was generated from the following file:

- SPA/DIK.h

4.5 Location< T, W > Struct Template Reference

```
#include <Maze.h>
```

Public Attributes

- T **row**
- T **col**
- W **weight** [4]

4.5.1 Detailed Description

template<typename T, typename W>
struct Location< T, W >

Location class that makes up the maze.

Template Parameters

T	The data type of row and column value of the maze.
W	The data type of the weights between adjacent locations.

The documentation for this struct was generated from the following file:

- Maze.h

4.6 Maze< T, W > Struct Template Reference

```
#include <Maze.h>
```

Public Member Functions

- Maze (T maxRow, T maxCol)
- ∼Maze ()
- void make ()
- void print () const
- Location< T, W > ∗ getAdjacentLoc (T row, T col, char dir) const

Public Attributes

- T **maxRow**
- T **maxColumn**
- Location< T, W > ∗∗ **location**

4.6.1 Detailed Description

template<typename T, typename W>
struct Maze< T, W >

A class that implements a maze with the location class. Using the Eller's algorithm to create a maze.

Template Parameters

T	The data type of row and column value of the maze.
W	The data type of the weights between adjacent locations.

4.6.2 Constructor & Destructor Documentation

4.6.2.1 Maze()

```
template<typename T , typename W >
Maze< T, W >::Maze (
             T maxRow,
             T maxCol )
```

Constructor for the Maze class.

Parameters

maxRow	Row size of the new maze.
maxCol	Column size of the new maze.

4.6.2.2 ∼Maze()

```
template<typename T , typename W >
Maze< T, W >::∼Maze
```

Destructor for the Maze class.

4.6.3 Member Function Documentation

4.6.3.1 getAdjacentLoc()

```
template<typename T , typename W >
Location< T, W > * Maze< T, W >::getAdjacentLoc (
             T row,
             T col,
             char dir ) const
```

Getter for adjacent location from current location and given direction.

Parameters

row	Row value of the current location.
col	Column value of the current location.
dir	Direction for the wanted adjacent location.

Returns

The adjacent location pointer.

4.6.3.2 make()

```
template<typename T , typename W >
void Maze< T, W >::make
```

Build the maze by opening some percentages of the walls in the maze by using the Eller's algorithm. For more information about the Eller's algorithm, visit `altair823's blog` and `The Buckblog`.

4.6.3.3 print()

```
template<typename T , typename W >
void Maze< T, W >::print
```

Print wall data of all locations in maze.

The documentation for this struct was generated from the following file:

- Maze.h

4.7 PriorityQueue< Key, Value > Class Template Reference

```
#include <PriorityQueue.h>
```

Public Member Functions

- PriorityQueue ()
- ∼PriorityQueue ()
- void push (Key key, Value &value)
- void pop ()
- Value top ()

4.7.1 Detailed Description

template<typename Key, typename Value>
class PriorityQueue< Key, Value >

Priority Queue class that is implemented with complete binary heap tree. Values in the queue is sorted using keys.

Template Parameters

Key	Data type for key.
Value	Data type for Value.

4.7.2 Constructor & Destructor Documentation

4.7.2.1 PriorityQueue()

```
template<typename Key , typename Value >
PriorityQueue< Key, Value >::PriorityQueue
```

Constructor a new priority queue.

4.7.2.2 ∼PriorityQueue()

```
template<typename Key , typename Value >
PriorityQueue< Key, Value >::∼PriorityQueue
```

Destructor a existing priority queue.

4.7.3 Member Function Documentation

4.7.3.1 pop()

```
template<typename Key , typename Value >
void PriorityQueue< Key, Value >::pop
```

Delete the data at the top.

4.7.3.2 push()

```
template<typename Key , typename Value >
void PriorityQueue< Key, Value >::push (
             Key key,
             Value & value )
```

Push a new key, value pair to the queue.

Parameters

key	A key of new data.
value	A value of new data.

4.7.3.3 top()

```
template<typename Key , typename Value >
Value PriorityQueue< Key, Value >::top
```

Return the value of data at the top.

Returns

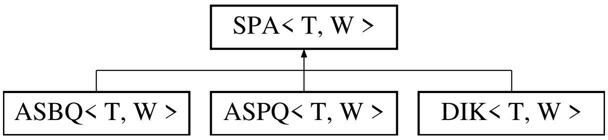
The value of data at the top.

The documentation for this class was generated from the following file:

- structure/PriorityQueue.h

4.8 SPA< T, W > Class Template Reference

Inheritance diagram for SPA< T, W >:



Public Member Functions

- SPA (Maze< T, W > &_maze)
- void setStart (T row, T column)
- void setEnd (T row, T column)
- virtual void findSP ()=0
- virtual W getShortestPathLength () const =0
- virtual std::string getTypeName () const =0

Protected Attributes

- Maze< T, W > & maze
- Location< T, W > ∗ end
- Location< T, W > ∗ start

4.8.1 Constructor & Destructor Documentation

4.8.1.1 SPA()

```
template<typename T , typename W >
SPA< T, W >::SPA (
            Maze< T, W > & _maze )  [inline], [explicit]
```

Constructor for SPA.

Parameters

_maze	The reference variable for the maze.

4.8.2 Member Function Documentation

4.8.2.1 findSP()

```
template<typename T , typename W >
virtual void SPA< T, W >::findSP ( )  [pure virtual]
```

Find the shortest path from the starting point to the ending point in the maze.

Implemented in ASBQ< T, W >, ASPQ< T, W >, and DIK< T, W >.

4.8.2.2 getShortestPathLength()

```
template<typename T , typename W >
virtual W SPA< T, W >::getShortestPathLength ( ) const  [pure virtual]
```

Getter for the length of the shortest path found.

Returns

The length of the shortest path found.

Implemented in ASBQ< T, W >, ASPQ< T, W >, and DIK< T, W >.

4.8.2.3 getTypeName()

```
template<typename T , typename W >
virtual std::string SPA< T, W >::getTypeName ( ) const  [pure virtual]
```

Getter for the name of the algorithm for finding the shortest path in a maze.

Returns

A name of the current algorithm.

Implemented in ASBQ< T, W >, ASPQ< T, W >, and DIK< T, W >.

4.8.2.4 setEnd()

```
template<typename T , typename W >
void SPA< T, W >::setEnd (
             T row,
             T column )  [inline]
```

Setter for the ending point of the maze.

Parameters

row	A row value of the ending point.
column	A column value of the ending point.

4.8.2.5 setStart()

```
template<typename T , typename W >
void SPA< T, W >::setStart (
            T row,
            T column )  [inline]
```

Setter for the starting point of the maze.

Parameters

row	A row value of the starting point.
column	A column value of the starting point.

4.8.3 Member Data Documentation

4.8.3.1 end

```
template<typename T , typename W >
Location<T, W>* SPA< T, W >::end  [protected]
```

The starting point for the shortest path of maze.

4.8.3.2 maze

```
template<typename T , typename W >
Maze<T, W>& SPA< T, W >::maze  [protected]
```

The maze reference variable.

4.8.3.3 start

```
template<typename T , typename W >
Location<T, W>* SPA< T, W >::start  [protected]
```

The destination point for the shortest path of maze.

The documentation for this class was generated from the following file:

- SPA/SPA.h

Chapter 5

File Documentation

5.1 Maze.h File Reference

Maze implementation file.

```
#include "pico/stdlib.h"
#include "hardware/structs/rosc.h"
```

Classes

- struct Location< T, W >
- struct Maze< T, W >

Macros

- #define **UP** 3
- #define **DOWN** 2
- #define **LEFT** 1
- #define **RIGHT** 0
- #define INF 10000
- #define WEIGHT_MAX 5
- #define WEIGHT_MIN 1
- #define ABS_MIN_WEIGHT(x, y) ((x->row - y->row) > 0 ? ((x->row - y->row) ∗ WEIGHT_MIN) : -((x->row - y->row) ∗ WEIGHT_MIN))
- #define GET_RAND_NUM(from, to) ((T) rand() % (to + 1 - from) + from)

5.1.1 Detailed Description

Maze implementation file.

Date

2022/02/17

Author

altair823

Version

1.0

5.1.2 Macro Definition Documentation

5.1.2.1 ABS_MIN_WEIGHT

```
#define ABS_MIN_WEIGHT(
              x,
              y ) ((x->row - y->row) > 0 ? ((x->row - y->row) * WEIGHT_MIN) : -((x->row -
y->row) * WEIGHT_MIN))
```

Calculate the absolute value of the distance between from x to y.

5.1.2.2 GET_RAND_NUM

```
#define GET_RAND_NUM(
              from,
              to ) ((T) rand() % (to + 1 - from) + from)
```

Generates a random number with a range between from and to.

5.1.2.3 INF

```
#define INF 10000
```

Pseudo-infinite value of the weight. This might be used to representing the closed wall.

5.1.2.4 WEIGHT_MAX

```
#define WEIGHT_MAX 5
```

The maximum value of the weight.

5.1.2.5 WEIGHT_MIN

```
#define WEIGHT_MIN 1
```

The minimum value of the weight.

5.2 Maze.h

Go to the documentation of this file.

```

#ifndef SPA_PICO_MAZE_H
#define SPA_PICO_MAZE_H

#include "pico/stdlib.h"
#include "hardware/structs/rosc.h"

/*
 * Directions for adjacent locations.
 */
#define UP 3
#define DOWN 2
#define LEFT 1
#define RIGHT 0


#define INF 10000
#define WEIGHT_MAX 5
#define WEIGHT_MIN 1

#define ABS_MIN_WEIGHT(x, y) ((x->row - y->row) > 0 ? ((x->row - y->row) * WEIGHT_MIN) : -((x->row - y->row) * WEIGHT_MIN))

#define GET_RAND_NUM(from, to) ((T) rand() % (to + 1 - from) + from)

template <typename T, typename W>
struct Location{
    T row;
    T col;
    W weight[4];
};

template <typename T, typename W>
struct Maze {
public:

    Maze(T maxRow, T maxCol);

    ~Maze();

    void make();

    void print() const;

    Location<T, W> *getAdjacentLoc(T row, T col, char dir) const;

    T maxRow;
    T maxColumn;
    Location<T, W> **location;
private:
    T *locationSet;
    T *nextLocationSet;
    T previouslyAssignedSetNumber;
    bool* existSetNumList;
    void openWall(T row, T column, char direction, W weight);
    void mergeWithRight(T row, T column);
    W generateWeight();
    void expandSetsVertical(T column);
    T getUnusedSetNumber();
};

template<typename T, typename W>
Maze<T, W>::Maze(T maxRow, T maxCol) {
    this->maxRow = maxRow;
    this->maxColumn = maxCol;
    location = new Location<T, W> *[maxCol];
    for (T tc = 0; tc < maxCol; tc++) {
        location[tc] = new Location<T, W>[maxRow];
        for (T tr = 0; tr < maxRow; ++tr) {
            location[tc][tr].row = tr;
            location[tc][tr].col = tc;

            location[tc][tr].weight[0] = INF;
            location[tc][tr].weight[1] = INF;
            location[tc][tr].weight[2] = INF;
            location[tc][tr].weight[3] = INF;
        }
    }

    previouslyAssignedSetNumber = 0;
    locationSet = new T[maxRow];
    nextLocationSet = new T[maxRow];
    existSetNumList = new bool[maxRow];

    // Seeding for random num.
    uint32_t random = 0x811c9dc5;
    uint8_t next_byte = 0;
    volatile uint32_t *rnd_reg = (uint32_t *) (ROSC_BASE + ROSC_RANDOMBIT_OFFSET);
    for (int i = 0; i < 16; i++) {
        for (int k = 0; k < 8; k++) {
            next_byte = (next_byte « 1) | (*rnd_reg & 1);
        }
        random ^= next_byte;
        random *= 0x01000193;
    }
    srand(random);
}

template<typename T, typename W>
Maze<T, W>::~Maze() {
    for (T i = 0; i < maxColumn; i++) {
        delete location[i];
    }
    delete[] location;
    delete[] locationSet;
    delete[] nextLocationSet;
    delete[] existSetNumList;
}

template<typename T, typename W>
void Maze<T, W>::make() {
    // Initial inserting. All cells in first row are inserted in different sets.
    for (T i = 0; i < maxRow; ++i) {
        locationSet[i] = i + 1;
    }
    for (T column = 0; column < maxColumn; ++column) {
        for (T r = 0; r < this->maxRow; ++r) {
            if (GET_RAND_NUM(0, 1) == 0) {
                this->mergeWithRight(r, column);
            }
        }
        expandSetsVertical(column);
        previouslyAssignedSetNumber = 0;
        for (T i = 0; i < maxRow; i++) {
            if (locationSet[i] == 0) {
                // Assign new set number to cell which does not have one.
                locationSet[i] = getUnusedSetNumber();
            }
        }
        // Last row, merge all cells that has different set value.
        if (column == maxColumn - 1) {
            for (T row = 0; row < maxRow - 1; ++row) {
                if (locationSet[row] != locationSet[row + 1]) {
                    mergeWithRight(row, column);
                }
            }
        }
    }
}

template<typename T, typename W>
void Maze<T, W>::print() const {
    for (T col = 0; col < maxColumn; col++) {
        for (T row = 0; row < maxRow; row++) {
            if (location[col][row].weight[UP] != INF) {
                std::cout « "U";
            } else { std::cout « "*"; }
            if (location[col][row].weight[DOWN] != INF) {
                std::cout « "D";
            } else { std::cout « "*"; }
            if (location[col][row].weight[LEFT] != INF) {
                std::cout « "L";
            } else { std::cout « "*"; }
            if (location[col][row].weight[RIGHT] != INF) {
                std::cout « "R";
            } else { std::cout « "*"; }
            std::cout « " ";
        }
        std::cout « "\n";
    }
}

template<typename T, typename W>
Location<T, W> *Maze<T, W>::getAdjacentLoc(T row, T col, char dir) const {
    if (location[col][row].weight[dir] >= INF) {
        return nullptr;
    }
    switch (dir) {
        case UP:
            if (col == 0) {
                return nullptr;
            } else {
                return &location[col - 1][row];
            }
        case DOWN:
            if (col == maxColumn - 1) {
                return nullptr;
            } else {
                return &location[col + 1][row];
            }
        case LEFT:
            if (row == 0) {
                return nullptr;
            } else {
                return &location[col][row - 1];
            }
        case RIGHT:
            if (row == maxRow - 1) {
                return nullptr;
            } else {
                return &location[col][row + 1];
            }
        default:
            std::cout « "wrong directiorn!";
            return nullptr;
    }
}

template<typename T, typename W>
void Maze<T, W>::openWall(T row, T column, char direction, W weight) {
    if ((row == 0 && direction == LEFT) ||
        (row == maxRow - 1 && direction == RIGHT) ||
        (column == 0 && direction == UP) ||
        (column == maxColumn - 1 && direction == DOWN)) {
        return;
    }
    // Open the wall in current cell location.
    location[column][row].weight[direction] = weight;

    // Open the wall in corresponding adjacent cell’s wall.
    switch (direction) {
        case UP:
            location[column - 1][row].weight[DOWN] = weight;
            break;
        case DOWN:
            location[column + 1][row].weight[UP] = weight;
            break;
        case LEFT:
            location[column][row - 1].weight[RIGHT] = weight;
            break;
        case RIGHT:
            location[column][row + 1].weight[LEFT] = weight;
            break;
        default:
            std::cout « "There is no adjacent cell in " « direction « " direction! (cell row: " « row
                      « ", cell col: " « column « ")" « std::endl;
            exit(1);
    }
}

template<typename T, typename W>
void Maze<T, W>::mergeWithRight(T row, T column) {
    // If the right side cell doesn’t exist, do nothing.
    if (row + 1 >= maxRow) {
        return;
    }
    // Groups two cells into the same set.
    T targetSetValue = locationSet[row + 1];
    T destSetValue = locationSet[row];
    for (T i = row; i < maxRow; i++) {
        if (locationSet[i] == targetSetValue) {
            locationSet[i] = destSetValue;
        }
    }
    // Open right side wall at the current cell.
    // This is accompanied by opening the left wall in the right cell.
    this->openWall(row, column, RIGHT, generateWeight());
}

template<typename T, typename W>
W Maze<T, W>::generateWeight() {
    W weight;
    // The Maximum weight is below (mean*2)
    do {
        weight = (W) GET_RAND_NUM(WEIGHT_MIN, WEIGHT_MAX);
    } while (weight <= WEIGHT_MIN || weight >= WEIGHT_MAX);
    return weight;
}

template<typename T, typename W>
void Maze<T, W>::expandSetsVertical(T column) {
    for (int i = 0; i < maxRow; i++) {
        existSetNumList[i] = false;
    }
    T SetStart = 0;
    T SetEnd = 0;
    T currentSet = 0;
    while (true) {
        for (T row = SetStart; row < maxRow; ++row) {
            // If new set is detected,
            if (locationSet[row] != 0 && currentSet == 0) {
                // set start point
                SetStart = row;
                currentSet = locationSet[row];
                existSetNumList[currentSet] = true;
                // delete set value because we don't need it anymore.
                locationSet[row] = 0;
                // If same set is detected,
            } else if (currentSet == locationSet[row]) {
                // just delete it.
                locationSet[row] = 0;
                // If different set is detected,
            } else if (currentSet != locationSet[row]) {
                // set end point before current row.
                SetEnd = row - 1;
                // But don't delete the set value.
                break;
            }
            if (row == maxRow - 1) {
                SetEnd = row;
                break;
            }
        }


        T expandCount = GET_RAND_NUM(1, (T) (SetEnd - SetStart + 1));
        for (; expandCount > 0; expandCount--) {
            T expandRow = GET_RAND_NUM(SetStart, SetEnd);
            // If new generated row value is already used, generate again.
            // This can be the bottleneck.
            if (nextLocationSet[expandRow] == currentSet) {
                expandCount++;
                continue;
                // Else, expand vertically.
            } else {
                nextLocationSet[expandRow] = currentSet;

                // Also merge two vertical cells in the real maze.

                if (column + 1 < maxColumn) {
                    this->openWall(expandRow, column, DOWN, generateWeight());
                }
            }
        }
        // End point is reached to the maximum, end the loop.
        // And update locationSet to nextLocationSet.
        if (SetEnd == maxRow - 1) {
            for (T i = 0; i < maxRow; i++) {
                locationSet[i] = nextLocationSet[i];
                nextLocationSet[i] = 0;
            }
            break;
        } else {
            SetStart = SetEnd + 1;
            currentSet = 0;
        }
    }
}

template<typename T, typename W>
T Maze<T, W>::getUnusedSetNumber() {
    // The number of sets is cannot over the maximum number of horizontal cells.
    for (T i = previouslyAssignedSetNumber + 1; i < maxRow + 1; i++) {
        // Find unused set number and assign it.
        if (existSetNumList[i] == false) {
            previouslyAssignedSetNumber = i;
            return i;
        }
    }
    exit(1);
}

#endif //SPA_PICO_MAZE_H
```

5.3 SPA/ASBQ.h File Reference

Implementation of A∗ algorithm using a bucket queue.

```
#include "../structure/BucketQueue.h"
#include "../Maze.h"
#include "SPA.h"
```

Classes

- class ASBQ< T, W >

5.3.1 Detailed Description

Implementation of A∗ algorithm using a bucket queue.

Date

2022/02/17

Author

altair823

Version

1.0

5.4 ASBQ.h

Go to the documentation of this file.

```

#ifndef SPA_PICO_ASBQ_H
#define SPA_PICO_ASBQ_H

#include "../structure/BucketQueue.h"
#include "../Maze.h"
#include "SPA.h"

template <typename T, typename W>
class ASBQ : public SPA<T, W> {
private:

    T maxRow, maxCol;

    W** distTable;

    BucketQueue<W, Location<T, W>*> bucketQueue;

    void UpdateDist(Location<T, W> *currentLoc);

public:
    ASBQ(T maxRow, T maxCol, Maze<T, W> &maze);

    ~ASBQ();

    void findSP() override;
    W getShortestPathLength() const override {return distTable[this->end->col][this->end->row];}
    [[nodiscard]] std::string getTypeName() const override {return "ASBQ  ";}
};

template<typename T, typename W>
ASBQ<T, W>::ASBQ(T _maxRow, T _maxCol, Maze<T, W> &_maze) :
            maxRow(_maxRow),
            maxCol(_maxCol),
            SPA<T, W>(_maze) {
    distTable = new W *[maxCol];
    for (T column = 0; column < maxCol; column++) {
        distTable[column] = new W[maxRow];
        for (T row = 0; row < maxRow; row++) {
            distTable[column][row] = INF;
        }
    }
}

template<typename T, typename W>
ASBQ<T, W>::~ASBQ() {
    for (T i = 0; i < maxCol; i++) {
        delete distTable[i];
    }
    delete distTable;
}

template<typename T, typename W>
void ASBQ<T, W>::findSP() {
    distTable[this->start->col][this->start->row] = 0;
    // Initially push the starting point to PQ.
    bucketQueue.push((W) (distTable[this->start->col][this->start->row] +
                          (ABS_MIN_WEIGHT(this->end, this->start)) +
                          (ABS_MIN_WEIGHT(this->end, this->start))),
                     this->start);
    auto currentLoc = this->start;
    while (currentLoc->row != this->end->row || currentLoc->col != this->end->col) {
        // Dequeue the closest location.
        // The distance of location from the starting point is used for only sorting.
        currentLoc = bucketQueue.top();
        bucketQueue.pop();
        // Update distance table for adjacent locations.
        UpdateDist(currentLoc);
    }
}

template<typename T, typename W>
void ASBQ<T, W>::UpdateDist(Location<T, W> *currentLoc) {
    for (char dir = 0; dir < 4; ++dir) {
        auto adjacent = this->maze.getAdjacentLoc(currentLoc->row, currentLoc->col, dir);
        // If there is adjacent location exists,
        // and its new distance is shorter then distance in the table, update it.
        if (adjacent != nullptr &&
            distTable[adjacent->col][adjacent->row] >
            currentLoc->weight[dir] + distTable[currentLoc->col][currentLoc->row]) {

            distTable[adjacent->col][adjacent->row] =
                    (W) (currentLoc->weight[dir] + distTable[currentLoc->col][currentLoc->row]);

            // Enqueue the new adjacent location which is updated just before.
            bucketQueue.push(
                    (W) (distTable[adjacent->col][adjacent->row] +
                         (ABS_MIN_WEIGHT(this->end, adjacent)) +
                         (ABS_MIN_WEIGHT(this->end, adjacent))),
                    adjacent);

        }
    }
}

#endif //SPA_PICO_ASBQ_H
```

5.5 SPA/ASPQ.h File Reference

Implementation of A∗ algorithm using a priority queue.

```
#include "../structure/PriorityQueue.h"
#include "../Maze.h"
#include "SPA.h"
```

Classes

- class ASPQ< T, W >

5.5.1 Detailed Description

Implementation of A∗ algorithm using a priority queue.

Date

2022/02/17

Author

altair823

Version

1.0

5.6 ASPQ.h

Go to the documentation of this file.

```

#ifndef SPA_PICO_ASPQ_H
#define SPA_PICO_ASPQ_H

#include "../structure/PriorityQueue.h"
#include "../Maze.h"
#include "SPA.h"

template <typename T, typename W>
class ASPQ : public SPA<T, W> {
private:

    T maxRow, maxCol;

    W** distTable;

    PriorityQueue<W, Location<T, W>*> adjacentLocQueue;

    void UpdateDist(Location<T, W> *currentLoc);

public:
    ASPQ(T maxRow, T maxCol, Maze<T, W> &maze);

    ~ASPQ();

    void findSP() override;
    W getShortestPathLength() const override {return distTable[this->end->col][this->end->row];}
    [[nodiscard]] std::string getTypeName() const override {return "ASPQ  ";}
};

template<typename T, typename W>
void ASPQ<T, W>::UpdateDist(Location<T, W> *currentLoc) {
    for (char dir = 0; dir < 4; ++dir) {
        auto adjacent = this->maze.getAdjacentLoc(currentLoc->row, currentLoc->col, dir);
        if (adjacent != nullptr &&
            distTable[adjacent->col][adjacent->row] >
            currentLoc->weight[dir] + distTable[currentLoc->col][currentLoc->row]) {

            distTable[adjacent->col][adjacent->row] =
                    (W) (currentLoc->weight[dir] + distTable[currentLoc->col][currentLoc->row]);

            // Enqueue the new adjacent location which is updated just before.
            adjacentLocQueue.push(
                    -(distTable[adjacent->col][adjacent->row] +
                      (ABS_MIN_WEIGHT(this->end, adjacent)) +
                      (ABS_MIN_WEIGHT(this->end, adjacent))),
                    adjacent);
        }
    }
}

template<typename T, typename W>
ASPQ<T, W>::ASPQ(T _maxRow, T _maxCol, Maze<T, W> &_maze):
maxRow(_maxRow),
maxCol(_maxCol),
SPA<T, W>(_maze) {
    distTable = new W *[maxCol];
    for (T column = 0; column < maxCol; column++) {
        distTable[column] = new W[maxRow];
        for (T row = 0; row < maxRow; row++) {
            distTable[column][row] = INF;
        }
    }
}

template<typename T, typename W>
ASPQ<T, W>::~ASPQ() {
    for (T i = 0; i < maxCol; i++) {
        delete distTable[i];
    }
    delete distTable;
}

template<typename T, typename W>
void ASPQ<T, W>::findSP() {
    distTable[this->start->col][this->start->row] = 0;
    // Initially push the starting point to PQ.
    adjacentLocQueue.push(-(distTable[this->start->col][this->start->row] +
                            (ABS_MIN_WEIGHT(this->end, this->start)) +
                            (ABS_MIN_WEIGHT(this->end, this->start))),
                          this->start);
    Location<T, W> *currentLoc = nullptr;
    while (currentLoc->row != this->end->row || currentLoc->col != this->end->col) {
        // Dequeue the closest location.
        // The distance of location from the starting point is used for only sorting.
        currentLoc = adjacentLocQueue.top();
        adjacentLocQueue.pop();
        // Update distance table for adjacent locations.
        UpdateDist(currentLoc);
    }
}

#endif //SPA_PICO_ASPQ_H
```

5.7 SPA/DIK.h File Reference

The naive implementation of the dijkstra algorithm.

```
#include "../Maze.h"
#include "SPA.h"
```

Classes

- class DIK< T, W >

Macros

- #define DEFAULT_LIST_CAP 10

5.7.1 Detailed Description

The naive implementation of the dijkstra algorithm.

Date

2022/02/17

Author

altair823

Version

1.0

5.7.2 Macro Definition Documentation

5.7.2.1 DEFAULT_LIST_CAP

```
#define DEFAULT_LIST_CAP 10
```

Initial default capacity of the list that contains all adjacent locations.

5.8 DIK.h

Go to the documentation of this file.

```

#ifndef SPA_PICO_DIK_H
#define SPA_PICO_DIK_H

#include "../Maze.h"
#include "SPA.h"

#define DEFAULT_LIST_CAP 10

template <typename T, typename W>
class DIK : public SPA<T, W> {
private:

    T maxRow, maxCol;

    W** distTable;

    bool** foundLocationSet;

    Location<T, W>** adjacentList;
    int adjacentListCap;
    int adjacentListTop;


    void UpdateDist(Location<T, W> *currentLoc);

public:
    DIK(T maxRow, T maxCol, Maze<T, W> &maze);

    ~DIK();

    void findSP() override;
    W getShortestPathLength() const override {return distTable[this->end->col][this->end->row];}
    [[nodiscard]] std::string getTypeName() const override {return "DIK   ";}
};

template<typename T, typename W>
void DIK<T, W>::UpdateDist(Location<T, W> *currentLoc) {
    // There are ways to improve performance at this point.
    // Such as data structure of adjacent vertices set.
    for (char dir = 0; dir < 4; dir++) {
        // For the adjacent currentLoc from all found locations,
        // if the adjacent currentLoc is not in the found currentLoc set,
        // calculate minimum distance and update if it is needed.
        // The edge vertices of maze are have nullptr for limits of maze size.
        auto adLoc = this->maze.getAdjacentLoc(currentLoc->row, currentLoc->col, dir);
        if (adLoc != nullptr && foundLocationSet[adLoc->col][adLoc->row] == false) {
            bool found = false;
            for (int p = 0; p < adjacentListTop; p++){
                if (adjacentList[p] == adLoc){
                    found = true;
                    break;
                }
            }
            if (!found) {
                if (adjacentListTop == adjacentListCap){
                    adjacentListCap *= 2;
                    auto t_adList = new Location<T, W>*[adjacentListCap];
                    for (int i = 0; i < adjacentListTop; i++){
                        t_adList[i] = adjacentList[i];
                    }
                    delete[] adjacentList;
                    adjacentList = t_adList;
                }
                adjacentList[adjacentListTop++] = adLoc;
            }
            if (distTable[adLoc->col][adLoc->row] >
                distTable[currentLoc->col][currentLoc->row] + currentLoc->weight[dir]) {
                distTable[adLoc->col][adLoc->row] =
                        distTable[currentLoc->col][currentLoc->row] + currentLoc->weight[dir];
            }
        }
    }
}

template<typename T, typename W>
DIK<T, W>::DIK(T _maxRow, T _maxCol, Maze<T, W> &_maze) :
maxRow(_maxRow),
maxCol(_maxCol),
adjacentList(new Location<T, W>*[DEFAULT_LIST_CAP]),
adjacentListCap(DEFAULT_LIST_CAP),
adjacentListTop(0),
SPA<T, W>(_maze){
    distTable = new W *[maxCol];
    foundLocationSet = new bool *[maxCol];
    for (T column = 0; column < maxCol; column++) {
        distTable[column] = new W[maxRow];
        foundLocationSet[column] = new bool[maxRow];
        for (T row = 0; row < maxRow; row++) {
            distTable[column][row] = INF;
            foundLocationSet[column][row] = false;
        }
    }
}

template<typename T, typename W>
DIK<T, W>::~DIK() {
    for (T i = 0; i < maxCol; i++) {
        delete[] distTable[i];
        delete[] foundLocationSet[i];
    }
    delete[] distTable;
    delete[] foundLocationSet;
    delete[] adjacentList;
}

template<typename T, typename W>
void DIK<T, W>::findSP() {
    // Insert starting point to found set.
    foundLocationSet[this->start->col][this->start->row] = true;

    distTable[this->start->col][this->start->row] = 0;

    auto currentLoc = this->start;
    int closestIndex = 0;

    // Finding the shortest path.
    while (currentLoc->row != this->end->row || currentLoc->col != this->end->col) {
        // 1. Update the distance to all vertices adjacent to the found location set.
        UpdateDist(currentLoc);

        // 2. Find the vertex which has minimum distance.
        int minDist = INF;
        for (T i = 0; i < adjacentListTop; ++i) {
            if (distTable[adjacentList[i]->col][adjacentList[i]->row] < minDist) {
                minDist = distTable[adjacentList[i]->col][adjacentList[i]->row];
                currentLoc = adjacentList[i];
                closestIndex = i;
            }
        }

        // 3. Insert that minimum vertex to the found location set.
        foundLocationSet[currentLoc->col][currentLoc->row] = true;

        // 4. Delete that minimum vertex from the adjacent location set.
        adjacentListTop--;
        for (int i = closestIndex; i < adjacentListTop; i++){
            adjacentList[i] = adjacentList[i+1];
        }
    }
}

#endif //SPA_PICO_DIK_H
```

5.9 SPA/SPA.h File Reference

Interface to an implementation of an algorithm that finds the shortest path in a given maze.

Classes

- class SPA< T, W >

5.9.1 Detailed Description

Interface to an implementation of an algorithm that finds the shortest path in a given maze.

Date

2022/02/17

Author

altair823

Version

1.0

5.10 SPA.h

Go to the documentation of this file.

```

#ifndef SPA_PICO_SPA_H
#define SPA_PICO_SPA_H

template <typename T, typename W>
class SPA {
protected:

    Maze<T, W> &maze;

    Location<T, W> *end;

    Location<T, W> *start;

public:
    explicit SPA(Maze<T, W> &_maze) : maze(_maze){}

    void setStart(T row, T column) {this->start = &(this->maze.location[column][row]);}

    void setEnd(T row, T column) {this->end = &(this->maze.location[column][row]);}

    virtual void findSP() = 0;

    virtual W getShortestPathLength() const = 0;

    [[nodiscard]] virtual std::string getTypeName() const = 0;
};


#endif //SPA_PICO_SPA_H
```

5.11 structure/BucketQueue.h File Reference

Bucket queue implementation.

Classes

- class BucketQueue< Key, Value >

Macros

- #define DEFAULT_B_QUEUE_CAP 10
- #define DEFAULT_BUCKET_CAP 4

5.11.1 Detailed Description

Bucket queue implementation.

Date

2022/02/17

Author

altair823

Version

1.0

5.11.2 Macro Definition Documentation

5.11.2.1 DEFAULT_B_QUEUE_CAP

```
#define DEFAULT_B_QUEUE_CAP 10
```

The initial default capacity of the bucket queue.

5.11.2.2 DEFAULT_BUCKET_CAP

```
#define DEFAULT_BUCKET_CAP 4
```

The initial default capacity of each bucket.

5.12 BucketQueue.h

Go to the documentation of this file.

```

#ifndef SPA_COMPARE_BUCKET_QUEUE_H
#define SPA_COMPARE_BUCKET_QUEUE_H

#define DEFAULT_B_QUEUE_CAP 10

#define DEFAULT_BUCKET_CAP 4

template <class Key, class Value>
class BucketQueue{
private:
    Value** bucketList;
    unsigned char *bucketTop;
    unsigned char *bucketCap;
    int bucketListSize;

    // Store the index of bucket that has the smallest key which is popped before.
    int minIndex = 0;
public:

    BucketQueue();

    ~BucketQueue();

    void push(Key key, Value &value);

    void pop();

    Value top();
};

template<class Key, class Value>
BucketQueue<Key, Value>::BucketQueue():
bucketList(new Value* [DEFAULT_B_QUEUE_CAP]),
bucketTop(new unsigned char[DEFAULT_B_QUEUE_CAP]),
bucketCap(new unsigned char[DEFAULT_B_QUEUE_CAP]){
    for (int i = 0; i < DEFAULT_B_QUEUE_CAP; i++){
        bucketCap[i] = DEFAULT_BUCKET_CAP;
        bucketList[i] = new Value[bucketCap[i]];
        bucketTop[i] = 0;
    }
    bucketListSize = DEFAULT_B_QUEUE_CAP;
}

template<class Key, class Value>
BucketQueue<Key, Value>::~BucketQueue() {
    for (int i = 0; i < bucketListSize; i++){
        if (bucketCap[i] > 0){
            delete[] bucketList[i];
        }
    }
    delete[] bucketList;
    delete[] bucketTop;
    delete[] bucketCap;
}

template<class Key, class Value>
void BucketQueue<Key, Value>::push(Key key, Value &value) {
    int newIndex = key;
    if (newIndex < minIndex){
        minIndex = newIndex;
    }
    if (bucketListSize <= newIndex && bucketListSize >= DEFAULT_B_QUEUE_CAP){
        auto t_queue = new Value*[(newIndex+1)*2];
        auto t_top = new unsigned char[(newIndex+1)*2];
        auto t_cap = new unsigned char[(newIndex+1)*2];
        for (int i = 0; i < (newIndex+1)*2; i++) {
            if (i < bucketListSize) {
                t_queue[i] = new Value[bucketCap[i]];
                for (int j = 0; j < bucketTop[i]; j++) {
                    t_queue[i][j] = bucketList[i][j];
                }
                t_top[i] = bucketTop[i];
                t_cap[i] = bucketCap[i];
                delete[] bucketList[i];
            } else{
                t_queue[i] = new Value[DEFAULT_BUCKET_CAP];
                t_top[i] = 0;
                t_cap[i] = DEFAULT_BUCKET_CAP;
            }
        }
        delete[] bucketList;
        delete[] bucketTop;
        delete[] bucketCap;
        bucketList = t_queue;
        bucketListSize = (newIndex+1)*2;
        bucketTop = t_top;
        bucketCap = t_cap;
    }
    if (bucketTop[newIndex] >= bucketCap[newIndex]){
        if (bucketCap[newIndex] < DEFAULT_BUCKET_CAP){
            bucketCap[newIndex] = DEFAULT_BUCKET_CAP;
        } else {
            bucketCap[newIndex] *= 2;
        }
        auto t_bucket = new Value[bucketCap[newIndex]];
        for (int i = 0; i < bucketTop[newIndex]; i++){
            t_bucket[i] = bucketList[newIndex][i];
        }
        delete[] bucketList[newIndex];
        bucketList[newIndex] = t_bucket;
    }
    bucketList[newIndex][bucketTop[newIndex]++] = value;
}

template<class Key, class Value>
void BucketQueue<Key, Value>::pop() {
    for (int i = minIndex; i < bucketListSize; ++i) {
        if (bucketTop[i] > 0) {
            minIndex = i;
            break;
        }
    }
    bucketTop[minIndex] = bucketTop[minIndex] > 0 ? bucketTop[minIndex] - 1 : 0;
    if (bucketTop[minIndex] == 0){
        delete[] bucketList[minIndex];
        bucketList[minIndex] = new Value[DEFAULT_BUCKET_CAP];
        bucketCap[minIndex] = DEFAULT_BUCKET_CAP;
    }
}

template<class Key, class Value>
Value BucketQueue<Key, Value>::top() {
    for (int i = minIndex; i < bucketListSize; ++i) {
        if (bucketTop[i] > 0) {
            minIndex = i;
            break;
        }
    }
    return bucketList[minIndex][bucketTop[minIndex] - 1];
}

#endif //SPA_COMPARE_BUCKET_QUEUE_H
```

5.13 structure/PriorityQueue.h File Reference

Priority queue implementation using complete binary heap tree.

Classes

- class PriorityQueue< Key, Value >

Macros

- #define DEFAULT_P_QUEUE_CAP 3
- #define SET_DATA(key, value, index)
- #define SWAP(indexA, indexB)

5.13.1 Detailed Description

Priority queue implementation using complete binary heap tree.

Date

2022/02/17

Author

altair823

Version

1.0

5.13.2 Macro Definition Documentation

5.13.2.1 DEFAULT_P_QUEUE_CAP

```
#define DEFAULT_P_QUEUE_CAP 3
```

The initial default capacity of data array in the queue.

5.13.2.2 SET_DATA

```
#define SET_DATA(
              key,
              value,
              index )
```

Value:

```
keyData[index] = key; \
valueData[index] = value
```

Set a key-value data to the array element in given index.

5.13.2.3 SWAP

```
#define SWAP(
                indexA,
                indexB )
```

Value:

```
auto tmpK = keyData[indexA]; \
keyData[indexA] = keyData[indexB]; \
keyData[indexB] = tmpK;     \
auto tmpV = valueData[indexA];    \
valueData[indexA] = valueData[indexB]; \
valueData[indexB] = tmpV
```

Swap the two data at index A and B.

5.14 PriorityQueue.h

Go to the documentation of this file.

```

#ifndef SPA_PICO_PRIORITY_QUEUE_H
#define SPA_PICO_PRIORITY_QUEUE_H

#define DEFAULT_P_QUEUE_CAP 3

#define SET_DATA(key, value, index) keyData[index] = key; \
                                    valueData[index] = value

#define SWAP(indexA, indexB)        auto tmpK = keyData[indexA]; \
                                    keyData[indexA] = keyData[indexB]; \
                                    keyData[indexB] = tmpK;     \
                                    auto tmpV = valueData[indexA];    \
                                    valueData[indexA] = valueData[indexB]; \
                                    valueData[indexB] = tmpV

template <typename Key, typename Value>
class PriorityQueue{
private:

    Key* keyData;

    Value* valueData;

    int capacity;

    int back;
public:
    PriorityQueue();

    ~PriorityQueue();

    void push(Key key, Value &value);

    void pop();

    Value top();
};

template<typename Key, typename Value>
PriorityQueue<Key, Value>::PriorityQueue():
keyData(new Key[DEFAULT_P_QUEUE_CAP]),
valueData(new Value[DEFAULT_P_QUEUE_CAP]),
capacity(DEFAULT_P_QUEUE_CAP),
back(0) {}

template<typename Key, typename Value>
PriorityQueue<Key, Value>::~PriorityQueue() {
    delete[] keyData;
    delete[] valueData;
}

template<typename Key, typename Value>
void PriorityQueue<Key, Value>::push(Key key, Value &value) {
    /* When the data array in the queue full, double the capacity of both arrays. */
    if (capacity == back) {
        capacity *= 2;

        auto tempKeyData = new Key[capacity];
        auto tempValueData = new Value [capacity];
        for (int i = 0; i < back; i++) {
            tempKeyData[i] = keyData[i];
            tempValueData[i] = valueData[i];
        }
        delete[] keyData;
        delete[] valueData;
        keyData = tempKeyData;
        valueData = tempValueData;
    }

    /* Appends new data to the end of the array. */
    SET_DATA(key, value, back);
    int parent = (back - 1) / 2;
    int child = back;

    /* Heapify all nodes. */
    while (parent >= 0 && keyData[parent] < keyData[child]) {
        SWAP(parent, child);
        child = parent;
        parent = (child - 1) / 2;
    }
    back++;
}

template<typename Key, typename Value>
void PriorityQueue<Key, Value>::pop() {
    if (back > 0) {
        back--;
        keyData[0] = keyData[back];
        valueData[0] = valueData[back];
        int parent = 0;
        int child = parent * 2 + 1;
        bool placed = false;

        /* Heapify all nodes. */
        while (!placed && child < back) {
            if (child < back - 1 && keyData[child] < keyData[child + 1])
                child += 1;
            /* Heapify complete. */
            if (keyData[parent] >= keyData[child])
                placed = true;
            else {
                SWAP(parent, child);
            }
            parent = child;
            child = parent * 2 + 1;
        }
    }
}

template<typename Key, typename Value>
Value PriorityQueue<Key, Value>::top() {
    if (back != 0){
        return valueData[0];
    } else {
        return nullptr;
    }
}

#endif //SPA_PICO_PRIORITY_QUEUE_H
```

Index