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:

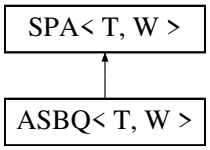
Chapter 4

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 >.

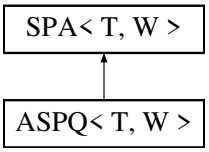
The documentation for this class was generated from the following file:

- 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.

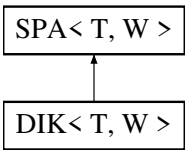
The documentation for this class was generated from the following file:

- 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

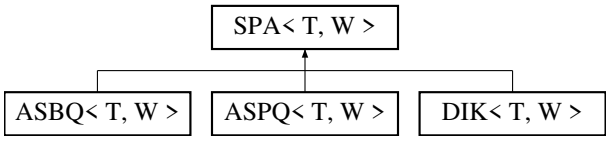
The value of data at the top.

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.

```
1
9 #ifndef SPA_PICO_MAZE_H
10 #define SPA_PICO_MAZE_H
11
12 #include "pico/stdlib.h"
13 #include "hardware/structs/rosc.h"
14
29 /*
30  * Directions for adjacent locations.
31  */
32 #define UP 3
33 #define DOWN 2
34 #define LEFT 1
35 #define RIGHT 0
36
37
38 #define INF 10000
39 #define WEIGHT_MAX 5
40 #define WEIGHT_MIN 1
41
45 #define ABS_MIN_WEIGHT(x, y) ((x->row - y->row) > 0 ? ((x->row - y->row) * WEIGHT_MIN) : -((x->row -
       y->row) * WEIGHT_MIN))
46
50 #define GET_RAND_NUM(from, to) ((T) rand() % (to + 1 - from) + from)
51
57 template <typename T, typename W>
58 struct Location{
59     T row;
60     T col;
61     W weight[4];
62 };
63
70 template <typename T, typename W>
71 struct Maze {
72 public:
73
79     Maze(T maxRow, T maxCol);
80
84     ~Maze();
85
91     void make();
92
96     void print() const;
97
105     Location<T, W> *getAdjacentLoc(T row, T col, char dir) const;
106
107     T maxRow;
108     T maxColumn;
109     Location<T, W> **location;
110 private:
111     T *locationSet;
112     T *nextLocationSet;
113     T previouslyAssignedSetNumber;
114     bool* existSetNumList;
115     void openWall(T row, T column, char direction, W weight);
116     void mergeWithRight(T row, T column);
117     W generateWeight();
118     void expandSetsVertical(T column);
119     T getUnusedSetNumber();
120 };
121
122 template<typename T, typename W>
123 Maze<T, W>::Maze(T maxRow, T maxCol) {
124     this->maxRow = maxRow;
125     this->maxColumn = maxCol;
126     location = new Location<T, W> *[maxCol];
127     for (T tc = 0; tc < maxCol; tc++) {
128         location[tc] = new Location<T, W>[maxRow];
129         for (T tr = 0; tr < maxRow; ++tr) {
130             location[tc][tr].row = tr;
131             location[tc][tr].col = tc;
132
133             location[tc][tr].weight[0] = INF;
134             location[tc][tr].weight[1] = INF;
135             location[tc][tr].weight[2] = INF;
136             location[tc][tr].weight[3] = INF;
137         }
138     }
139
140     previouslyAssignedSetNumber = 0;
141     locationSet = new T[maxRow];
142     nextLocationSet = new T[maxRow];
```

```
143     existSetNumList = new bool[maxRow];
144
145     // Seeding for random num.
146     uint32_t random = 0x811c9dc5;
147     uint8_t next_byte = 0;
148     volatile uint32_t *rnd_reg = (uint32_t *) (ROSC_BASE + ROSC_RANDOMBIT_OFFSET);
149     for (int i = 0; i < 16; i++) {
150         for (int k = 0; k < 8; k++) {
151             next_byte = (next_byte << 1) | (*rnd_reg & 1);
152         }
153         random ^= next_byte;
154         random *= 0x01000193;
155     }
156     srand(random);
157 }
158
159 template<typename T, typename W>
160 Maze<T, W>::~Maze() {
161     for (T i = 0; i < maxColumn; i++) {
162         delete location[i];
163     }
164     delete[] location;
165     delete[] locationSet;
166     delete[] nextLocationSet;
167     delete[] existSetNumList;
168 }
169
170 template<typename T, typename W>
171 void Maze<T, W>::make() {
172     // Initial inserting. All cells in first row are inserted in different sets.
173     for (T i = 0; i < maxRow; ++i) {
174         locationSet[i] = i + 1;
175     }
176     for (T column = 0; column < maxColumn; ++column) {
177         for (T r = 0; r < this->maxRow; ++r) {
178             if (GET_RAND_NUM(0, 1) == 0) {
179                 this->mergeWithRight(r, column);
180             }
181         }
182         expandSetsVertical(column);
183         previouslyAssignedSetNumber = 0;
184         for (T i = 0; i < maxRow; i++) {
185             if (locationSet[i] == 0) {
186                 // Assign new set number to cell which does not have one.
187                 locationSet[i] = getUnusedSetNumber();
188             }
189         }
190         // Last row, merge all cells that has different set value.
191         if (column == maxColumn - 1) {
192             for (T row = 0; row < maxRow - 1; ++row) {
193                 if (locationSet[row] != locationSet[row + 1]) {
194                     mergeWithRight(row, column);
195                 }
196             }
197         }
198     }
199 }
200
201 template<typename T, typename W>
202 void Maze<T, W>::print() const {
203     for (T col = 0; col < maxColumn; col++) {
204         for (T row = 0; row < maxRow; row++) {
205             if (location[col][row].weight[UP] != INF) {
206                 std::cout << "U";
207             } else { std::cout << "*"; }
208             if (location[col][row].weight[DOWN] != INF) {
209                 std::cout << "D";
210             } else { std::cout << "*"; }
211             if (location[col][row].weight[LEFT] != INF) {
212                 std::cout << "L";
213             } else { std::cout << "*"; }
214             if (location[col][row].weight[RIGHT] != INF) {
215                 std::cout << "R";
216             } else { std::cout << "*"; }
217             std::cout << " ";
218         }
219         std::cout << "\n";
220     }
221 }
222
223 template<typename T, typename W>
224 Location<T, W> *Maze<T, W>::getAdjacentLoc(T row, T col, char dir) const {
225     if (location[col][row].weight[dir] >= INF) {
226         return nullptr;
227     }
228     switch (dir) {
229         case UP:
```

```
230             if (col == 0) {
231                 return nullptr;
232             } else {
233                 return &location[col - 1][row];
234             }
235         case DOWN:
236             if (col == maxColumn - 1) {
237                 return nullptr;
238             } else {
239                 return &location[col + 1][row];
240             }
241         case LEFT:
242             if (row == 0) {
243                 return nullptr;
244             } else {
245                 return &location[col][row - 1];
246             }
247         case RIGHT:
248             if (row == maxRow - 1) {
249                 return nullptr;
250             } else {
251                 return &location[col][row + 1];
252             }
253         default:
254             std::cout « "wrong directiorn!";
255             return nullptr;
256     }
257 }
258
259 template<typename T, typename W>
260 void Maze<T, W>::openWall(T row, T column, char direction, W weight) {
261     if ((row == 0 && direction == LEFT) ||
262         (row == maxRow - 1 && direction == RIGHT) ||
263         (column == 0 && direction == UP) ||
264         (column == maxColumn - 1 && direction == DOWN)) {
265         return;
266     }
267     // Open the wall in current cell location.
268     location[column][row].weight[direction] = weight;
269
270     // Open the wall in corresponding adjacent cell's wall.
271     switch (direction) {
272         case UP:
273             location[column - 1][row].weight[DOWN] = weight;
274             break;
275         case DOWN:
276             location[column + 1][row].weight[UP] = weight;
277             break;
278         case LEFT:
279             location[column][row - 1].weight[RIGHT] = weight;
280             break;
281         case RIGHT:
282             location[column][row + 1].weight[LEFT] = weight;
283             break;
284         default:
285             std::cout « "There is no adjacent cell in " « direction « " direction! (cell row: " « row
286                      « ", cell col: " « column « ")" « std::endl;
287             exit(1);
288     }
289 }
290
291 template<typename T, typename W>
292 void Maze<T, W>::mergeWithRight(T row, T column) {
293     // If the right side cell doesn't exist, do nothing.
294     if (row + 1 >= maxRow) {
295         return;
296     }
297     // Groups two cells into the same set.
298     T targetSetValue = locationSet[row + 1];
299     T destSetValue = locationSet[row];
300     for (T i = row; i < maxRow; i++) {
301         if (locationSet[i] == targetSetValue) {
302             locationSet[i] = destSetValue;
303         }
304     }
305     // Open right side wall at the current cell.
306     // This is accompanied by opening the left wall in the right cell.
307     this->openWall(row, column, RIGHT, generateWeight());
308 }
309
310 template<typename T, typename W>
311 W Maze<T, W>::generateWeight() {
312     W weight;
313     // The Maximum weight is below (mean*2)
314     do {
315         weight = (W) GET_RAND_NUM(WEIGHT_MIN, WEIGHT_MAX);
316     } while (weight <= WEIGHT_MIN || weight >= WEIGHT_MAX);
```

```
317     return weight;
318 }
319
320 template<typename T, typename W>
321 void Maze<T, W>::expandSetsVertical(T column) {
322     for (int i = 0; i < maxRow; i++) {
323         existSetNumList[i] = false;
324     }
325     T SetStart = 0;
326     T SetEnd = 0;
327     T currentSet = 0;
328     while (true) {
329         for (T row = SetStart; row < maxRow; ++row) {
330             // If new set is detected,
331             if (locationSet[row] != 0 && currentSet == 0) {
332                 // set start point
333                 SetStart = row;
334                 currentSet = locationSet[row];
335                 existSetNumList[currentSet] = true;
336                 // delete set value because we don't need it anymore.
337                 locationSet[row] = 0;
338                 // If same set is detected,
339             } else if (currentSet == locationSet[row]) {
340                 // just delete it.
341                 locationSet[row] = 0;
342                 // If different set is detected,
343             } else if (currentSet != locationSet[row]) {
344                 // set end point before current row.
345                 SetEnd = row - 1;
346                 // But don't delete the set value.
347                 break;
348             }
349             if (row == maxRow - 1) {
350                 SetEnd = row;
351                 break;
352             }
353         }
354
355
356         T expandCount = GET_RAND_NUM(1, (T) (SetEnd - SetStart + 1));
357         for (; expandCount > 0; expandCount--) {
358             T expandRow = GET_RAND_NUM(SetStart, SetEnd);
359             // If new generated row value is already used, generate again.
360             // This can be the bottleneck.
361             if (nextLocationSet[expandRow] == currentSet) {
362                 expandCount++;
363                 continue;
364                 // Else, expand vertically.
365             } else {
366                 nextLocationSet[expandRow] = currentSet;
367
368                 // Also merge two vertical cells in the real maze.
369
370                 if (column + 1 < maxColumn) {
371                     this->openWall(expandRow, column, DOWN, generateWeight());
372                 }
373             }
374         }
375         // End point is reached to the maximum, end the loop.
376         // And update locationSet to nextLocationSet.
377         if (SetEnd == maxRow - 1) {
378             for (T i = 0; i < maxRow; i++) {
379                 locationSet[i] = nextLocationSet[i];
380                 nextLocationSet[i] = 0;
381             }
382             break;
383         } else {
384             SetStart = SetEnd + 1;
385             currentSet = 0;
386         }
387     }
388 }
389
390 template<typename T, typename W>
391 T Maze<T, W>::getUnusedSetNumber() {
392     // The number of sets is cannot over the maximum number of horizontal cells.
393     for (T i = previouslyAssignedSetNumber + 1; i < maxRow + 1; i++) {
394         // Find unused set number and assign it.
395         if (existSetNumList[i] == false) {
396             previouslyAssignedSetNumber = i;
397             return i;
398         }
399     }
400     exit(1);
401 }
402
403 #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.

```
1
9  #ifndef SPA_PICO_ASBQ_H
10 #define SPA_PICO_ASBQ_H
11
12 #include "../structure/BucketQueue.h"
13 #include "../Maze.h"
14 #include "SPA.h"
15
22 template <typename T, typename W>
23 class ASBQ : public SPA<T, W> {
24 private:
25
29     T maxRow, maxCol;
30
34     W** distTable;
35
41     BucketQueue<W, Location<T, W>*> bucketQueue;
42
47     void UpdateDist(Location<T, W> *currentLoc);
48
49 public:
56     ASBQ(T maxRow, T maxCol, Maze<T, W> &maze);
57
61     ~ASBQ();
62
63     void findSP() override;
64     W getShortestPathLength() const override {return distTable[this->end->col][this->end->row];}
65     [[nodiscard]] std::string getTypeName() const override {return "ASBQ  ";}
```

```
66 };
67
68 template<typename T, typename W>
69 ASBQ<T, W>::ASBQ(T _maxRow, T _maxCol, Maze<T, W> &_maze) :
70             maxRow(_maxRow),
71             maxCol(_maxCol),
72             SPA<T, W>(_maze) {
73     distTable = new W *[maxCol];
74     for (T column = 0; column < maxCol; column++) {
75         distTable[column] = new W[maxRow];
76         for (T row = 0; row < maxRow; row++) {
77             distTable[column][row] = INF;
78         }
79     }
80 }
81
82 template<typename T, typename W>
83 ASBQ<T, W>::~ASBQ() {
84     for (T i = 0; i < maxCol; i++) {
85         delete distTable[i];
86     }
87     delete distTable;
88 }
89
90 template<typename T, typename W>
91 void ASBQ<T, W>::findSP() {
92     distTable[this->start->col][this->start->row] = 0;
93     // Initially push the starting point to PQ.
94     bucketQueue.push((W) (distTable[this->start->col][this->start->row] +
95                           (ABS_MIN_WEIGHT(this->end, this->start)) +
96                           (ABS_MIN_WEIGHT(this->end, this->start))),
97                      this->start);
98     auto currentLoc = this->start;
99     while (currentLoc->row != this->end->row || currentLoc->col != this->end->col) {
100         // Dequeue the closest location.
101         // The distance of location from the starting point is used for only sorting.
102         currentLoc = bucketQueue.top();
103         bucketQueue.pop();
104         // Update distance table for adjacent locations.
105         UpdateDist(currentLoc);
106     }
107 }
108
109 template<typename T, typename W>
110 void ASBQ<T, W>::UpdateDist(Location<T, W> *currentLoc) {
111     for (char dir = 0; dir < 4; ++dir) {
112         auto adjacent = this->maze.getAdjacentLoc(currentLoc->row, currentLoc->col, dir);
113         // If there is adjacent location exists,
114         // and its new distance is shorter then distance in the table, update it.
115         if (adjacent != nullptr &&
116             distTable[adjacent->col][adjacent->row] >
117             currentLoc->weight[dir] + distTable[currentLoc->col][currentLoc->row]) {
118
119             distTable[adjacent->col][adjacent->row] =
120                     (W) (currentLoc->weight[dir] + distTable[currentLoc->col][currentLoc->row]);
121
122             // Enqueue the new adjacent location which is updated just before.
123             bucketQueue.push(
124                     (W) (distTable[adjacent->col][adjacent->row] +
125                          (ABS_MIN_WEIGHT(this->end, adjacent)) +
126                          (ABS_MIN_WEIGHT(this->end, adjacent))),
127                     adjacent);
128
129         }
130     }
131 }
132
133 #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.

```
1
9 #ifndef SPA_PICO_ASPQ_H
10 #define SPA_PICO_ASPQ_H
11
12 #include "../structure/PriorityQueue.h"
13 #include "../Maze.h"
14 #include "SPA.h"
15
22 template <typename T, typename W>
23 class ASPQ : public SPA<T, W> {
24 private:
25
29     T maxRow, maxCol;
30
34     W** distTable;
35
41     PriorityQueue<W, Location<T, W>*> adjacentLocQueue;
42
47     void UpdateDist(Location<T, W> *currentLoc);
48
49 public:
56     ASPQ(T maxRow, T maxCol, Maze<T, W> &maze);
57
61     ~ASPQ();
62
63     void findSP() override;
64     W getShortestPathLength() const override {return distTable[this->end->col][this->end->row];}
65     [[nodiscard]] std::string getTypeName() const override {return "ASPQ  ";}
66 };
67
68 template<typename T, typename W>
69 void ASPQ<T, W>::UpdateDist(Location<T, W> *currentLoc) {
70     for (char dir = 0; dir < 4; ++dir) {
71         auto adjacent = this->maze.getAdjacentLoc(currentLoc->row, currentLoc->col, dir);
72         if (adjacent != nullptr &&
73             distTable[adjacent->col][adjacent->row] >
74             currentLoc->weight[dir] + distTable[currentLoc->col][currentLoc->row]) {
75
76             distTable[adjacent->col][adjacent->row] =
77                     (W) (currentLoc->weight[dir] + distTable[currentLoc->col][currentLoc->row]);
78
79             // Enqueue the new adjacent location which is updated just before.
80             adjacentLocQueue.push(
81                     -(distTable[adjacent->col][adjacent->row] +
```

```
82                      (ABS_MIN_WEIGHT(this->end, adjacent)) +
83                      (ABS_MIN_WEIGHT(this->end, adjacent))),
84                    adjacent);
85         }
86     }
87 }
88
89 template<typename T, typename W>
90 ASPQ<T, W>::ASPQ(T _maxRow, T _maxCol, Maze<T, W> &_maze):
91 maxRow(_maxRow),
92 maxCol(_maxCol),
93 SPA<T, W>(_maze) {
94     distTable = new W *[maxCol];
95     for (T column = 0; column < maxCol; column++) {
96         distTable[column] = new W[maxRow];
97         for (T row = 0; row < maxRow; row++) {
98             distTable[column][row] = INF;
99         }
100     }
101 }
102
103 template<typename T, typename W>
104 ASPQ<T, W>::~ASPQ() {
105     for (T i = 0; i < maxCol; i++) {
106         delete distTable[i];
107     }
108     delete distTable;
109 }
110
111 template<typename T, typename W>
112 void ASPQ<T, W>::findSP() {
113     distTable[this->start->col][this->start->row] = 0;
114     // Initially push the starting point to PQ.
115     adjacentLocQueue.push(-(distTable[this->start->col][this->start->row] +
116                             (ABS_MIN_WEIGHT(this->end, this->start)) +
117                             (ABS_MIN_WEIGHT(this->end, this->start))),
118                           this->start);
119     Location<T, W> *currentLoc = nullptr;
120     while (currentLoc->row != this->end->row || currentLoc->col != this->end->col) {
121         // Dequeue the closest location.
122         // The distance of location from the starting point is used for only sorting.
123         currentLoc = adjacentLocQueue.top();
124         adjacentLocQueue.pop();
125         // Update distance table for adjacent locations.
126         UpdateDist(currentLoc);
127     }
128 }
129
130 #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.

```
1
9 #ifndef SPA_PICO_DIK_H
10 #define SPA_PICO_DIK_H
11
12 #include "../Maze.h"
13 #include "SPA.h"
14
19 #define DEFAULT_LIST_CAP 10
20
27 template <typename T, typename W>
28 class DIK : public SPA<T, W> {
29 private:
30
34     T maxRow, maxCol;
35
39     W** distTable;
40
44     bool** foundLocationSet;
45
49     Location<T, W>** adjacentList;
50     int adjacentListCap;
51     int adjacentListTop;
52
53
58     void UpdateDist(Location<T, W> *currentLoc);
59
60 public:
67     DIK(T maxRow, T maxCol, Maze<T, W> &maze);
68
72     ~DIK();
73
74     void findSP() override;
```

```
75      W getShortestPathLength() const override {return distTable[this->end->col][this->end->row];}
76      [[nodiscard]] std::string getTypeName() const override {return "DIK   ";}
77  };
78
79  template<typename T, typename W>
80  void DIK<T, W>::UpdateDist(Location<T, W> *currentLoc) {
81      // There are ways to improve performance at this point.
82      // Such as data structure of adjacent vertices set.
83      for (char dir = 0; dir < 4; dir++) {
84          // For the adjacent currentLoc from all found locations,
85          // if the adjacent currentLoc is not in the found currentLoc set,
86          // calculate minimum distance and update if it is needed.
87          // The edge vertices of maze are have nullptr for limits of maze size.
88          auto adLoc = this->maze.getAdjacentLoc(currentLoc->row, currentLoc->col, dir);
89          if (adLoc != nullptr && foundLocationSet[adLoc->col][adLoc->row] == false) {
90              bool found = false;
91              for (int p = 0; p < adjacentListTop; p++){
92                  if (adjacentList[p] == adLoc){
93                      found = true;
94                      break;
95                  }
96              }
97              if (!found) {
98                  if (adjacentListTop == adjacentListCap){
99                      adjacentListCap *= 2;
100                     auto t_adList = new Location<T, W>*[adjacentListCap];
101                     for (int i = 0; i < adjacentListTop; i++){
102                         t_adList[i] = adjacentList[i];
103                     }
104                     delete[] adjacentList;
105                     adjacentList = t_adList;
106                 }
107                 adjacentList[adjacentListTop++] = adLoc;
108             }
109             if (distTable[adLoc->col][adLoc->row] >
110                 distTable[currentLoc->col][currentLoc->row] + currentLoc->weight[dir]) {
111                 distTable[adLoc->col][adLoc->row] =
112                         distTable[currentLoc->col][currentLoc->row] + currentLoc->weight[dir];
113             }
114         }
115     }
116 }
117
118 template<typename T, typename W>
119 DIK<T, W>::DIK(T _maxRow, T _maxCol, Maze<T, W> &_maze) :
120 maxRow(_maxRow),
121 maxCol(_maxCol),
122 adjacentList(new Location<T, W>*[DEFAULT_LIST_CAP]),
123 adjacentListCap(DEFAULT_LIST_CAP),
124 adjacentListTop(0),
125 SPA<T, W>(_maze){
126     distTable = new W *[maxCol];
127     foundLocationSet = new bool *[maxCol];
128     for (T column = 0; column < maxCol; column++) {
129         distTable[column] = new W[maxRow];
130         foundLocationSet[column] = new bool[maxRow];
131         for (T row = 0; row < maxRow; row++) {
132             distTable[column][row] = INF;
133             foundLocationSet[column][row] = false;
134         }
135     }
136 }
137
138 template<typename T, typename W>
139 DIK<T, W>::~DIK() {
140     for (T i = 0; i < maxCol; i++) {
141         delete[] distTable[i];
142         delete[] foundLocationSet[i];
143     }
144     delete[] distTable;
145     delete[] foundLocationSet;
146     delete[] adjacentList;
147 }
148
149 template<typename T, typename W>
150 void DIK<T, W>::findSP() {
151     // Insert starting point to found set.
152     foundLocationSet[this->start->col][this->start->row] = true;
153
154     distTable[this->start->col][this->start->row] = 0;
155
156     auto currentLoc = this->start;
157     int closestIndex = 0;
158
159     // Finding the shortest path.
160     while (currentLoc->row != this->end->row || currentLoc->col != this->end->col) {
161         // 1. Update the distance to all vertices adjacent to the found location set.
```

```
162         UpdateDist(currentLoc);
163
164         // 2. Find the vertex which has minimum distance.
165         int minDist = INF;
166         for (T i = 0; i < adjacentListTop; ++i) {
167             if (distTable[adjacentList[i]->col][adjacentList[i]->row] < minDist) {
168                 minDist = distTable[adjacentList[i]->col][adjacentList[i]->row];
169                 currentLoc = adjacentList[i];
170                 closestIndex = i;
171             }
172         }
173
174         // 3. Insert that minimum vertex to the found location set.
175         foundLocationSet[currentLoc->col][currentLoc->row] = true;
176
177         // 4. Delete that minimum vertex from the adjacent location set.
178         adjacentListTop--;
179         for (int i = closestIndex; i < adjacentListTop; i++){
180             adjacentList[i] = adjacentList[i+1];
181         }
182     }
183 }
184
185 #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.

```
1
9 #ifndef SPA_PICO_SPA_H
10 #define SPA_PICO_SPA_H
11
12 template <typename T, typename W>
13 class SPA {
14 protected:
15
19     Maze<T, W> &maze;
20
24     Location<T, W> *end;
25
29     Location<T, W> *start;
30
31 public:
36     explicit SPA(Maze<T, W> &_maze) : maze(_maze){}
37
43     void setStart(T row, T column) {this->start = &(this->maze.location[column][row]);}
44
50     void setEnd(T row, T column) {this->end = &(this->maze.location[column][row]);}
51
55     virtual void findSP() = 0;
56
61     virtual W getShortestPathLength() const = 0;
62
67     [[nodiscard]] virtual std::string getTypeName() const = 0;
68 };
69
70
71 #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.

```
1
9  #ifndef SPA_COMPARE_BUCKET_QUEUE_H
10 #define SPA_COMPARE_BUCKET_QUEUE_H
11
16 #define DEFAULT_B_QUEUE_CAP 10
17
22 #define DEFAULT_BUCKET_CAP 4
23
30 template <class Key, class Value>
31 class BucketQueue{
32 private:
33     Value** bucketList;
34     unsigned char *bucketTop;
35     unsigned char *bucketCap;
36     int bucketListSize;
37
38     // Store the index of bucket that has the smallest key which is popped before.
39     int minIndex = 0;
40 public:
41
45     BucketQueue();
46
50     ~BucketQueue();
51
57     void push(Key key, Value &value);
58
62     void pop();
63
68     Value top();
69 };
70
71 template<class Key, class Value>
72 BucketQueue<Key, Value>::BucketQueue():
73 bucketList(new Value* [DEFAULT_B_QUEUE_CAP]),
74 bucketTop(new unsigned char[DEFAULT_B_QUEUE_CAP]),
75 bucketCap(new unsigned char[DEFAULT_B_QUEUE_CAP]){
76     for (int i = 0; i < DEFAULT_B_QUEUE_CAP; i++){
77         bucketCap[i] = DEFAULT_BUCKET_CAP;
78         bucketList[i] = new Value[bucketCap[i]];
79         bucketTop[i] = 0;
80     }
81     bucketListSize = DEFAULT_B_QUEUE_CAP;
82 }
83
84 template<class Key, class Value>
85 BucketQueue<Key, Value>::~BucketQueue() {
86     for (int i = 0; i < bucketListSize; i++){
87         if (bucketCap[i] > 0){
88             delete[] bucketList[i];
```

```
89          }
90      }
91      delete[] bucketList;
92      delete[] bucketTop;
93      delete[] bucketCap;
94  }
95
96  template<class Key, class Value>
97  void BucketQueue<Key, Value>::push(Key key, Value &value) {
98      int newIndex = key;
99      if (newIndex < minIndex){
100         minIndex = newIndex;
101     }
102     if (bucketListSize <= newIndex && bucketListSize >= DEFAULT_B_QUEUE_CAP){
103         auto t_queue = new Value*[(newIndex+1)*2];
104         auto t_top = new unsigned char[(newIndex+1)*2];
105         auto t_cap = new unsigned char[(newIndex+1)*2];
106         for (int i = 0; i < (newIndex+1)*2; i++) {
107             if (i < bucketListSize) {
108                 t_queue[i] = new Value[bucketCap[i]];
109                 for (int j = 0; j < bucketTop[i]; j++) {
110                     t_queue[i][j] = bucketList[i][j];
111                 }
112                 t_top[i] = bucketTop[i];
113                 t_cap[i] = bucketCap[i];
114                 delete[] bucketList[i];
115             } else{
116                 t_queue[i] = new Value[DEFAULT_BUCKET_CAP];
117                 t_top[i] = 0;
118                 t_cap[i] = DEFAULT_BUCKET_CAP;
119             }
120         }
121         delete[] bucketList;
122         delete[] bucketTop;
123         delete[] bucketCap;
124         bucketList = t_queue;
125         bucketListSize = (newIndex+1)*2;
126         bucketTop = t_top;
127         bucketCap = t_cap;
128     }
129     if (bucketTop[newIndex] >= bucketCap[newIndex]){
130         if (bucketCap[newIndex] < DEFAULT_BUCKET_CAP){
131             bucketCap[newIndex] = DEFAULT_BUCKET_CAP;
132         } else {
133             bucketCap[newIndex] *= 2;
134         }
135         auto t_bucket = new Value[bucketCap[newIndex]];
136         for (int i = 0; i < bucketTop[newIndex]; i++){
137             t_bucket[i] = bucketList[newIndex][i];
138         }
139         delete[] bucketList[newIndex];
140         bucketList[newIndex] = t_bucket;
141     }
142     bucketList[newIndex][bucketTop[newIndex]++] = value;
143 }
144
145 template<class Key, class Value>
146 void BucketQueue<Key, Value>::pop() {
147     for (int i = minIndex; i < bucketListSize; ++i) {
148         if (bucketTop[i] > 0) {
149             minIndex = i;
150             break;
151         }
152     }
153     bucketTop[minIndex] = bucketTop[minIndex] > 0 ? bucketTop[minIndex] - 1 : 0;
154     if (bucketTop[minIndex] == 0){
155         delete[] bucketList[minIndex];
156         bucketList[minIndex] = new Value[DEFAULT_BUCKET_CAP];
157         bucketCap[minIndex] = DEFAULT_BUCKET_CAP;
158     }
159 }
160
161 template<class Key, class Value>
162 Value BucketQueue<Key, Value>::top() {
163     for (int i = minIndex; i < bucketListSize; ++i) {
164         if (bucketTop[i] > 0) {
165             minIndex = i;
166             break;
167         }
168     }
169     return bucketList[minIndex][bucketTop[minIndex] - 1];
170 }
171
172 #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.

```
1
9  #ifndef SPA_PICO_PRIORITY_QUEUE_H
10 #define SPA_PICO_PRIORITY_QUEUE_H
11
16 #define DEFAULT_P_QUEUE_CAP 3
17
22 #define SET_DATA(key, value, index) keyData[index] = key; \
23                                     valueData[index] = value
24
29 #define SWAP(indexA, indexB)        auto tmpK = keyData[indexA]; \
30                                     keyData[indexA] = keyData[indexB]; \
31                                     keyData[indexB] = tmpK;      \
32                                     auto tmpV = valueData[indexA];    \
33                                     valueData[indexA] = valueData[indexB]; \
34                                     valueData[indexB] = tmpV
35
42 template <typename Key, typename Value>
43 class PriorityQueue{
44 private:
45
49     Key* keyData;
50
54     Value* valueData;
55
59     int capacity;
60
64     int back;
65 public:
69     PriorityQueue();
70
74     ~PriorityQueue();
75
81     void push(Key key, Value &value);
82
86     void pop();
87
92     Value top();
93 };
94
95 template<typename Key, typename Value>
96 PriorityQueue<Key, Value>::PriorityQueue():
97 keyData(new Key[DEFAULT_P_QUEUE_CAP]),
98 valueData(new Value[DEFAULT_P_QUEUE_CAP]),
99 capacity(DEFAULT_P_QUEUE_CAP),
100 back(0) {}
101
102 template<typename Key, typename Value>
103 PriorityQueue<Key, Value>::~PriorityQueue() {
104     delete[] keyData;
105     delete[] valueData;
106 }
107
108 template<typename Key, typename Value>
109 void PriorityQueue<Key, Value>::push(Key key, Value &value) {
110     /* When the data array in the queue full, double the capacity of both arrays. */
111     if (capacity == back) {
112         capacity *= 2;
```

```
113
114         auto tempKeyData = new Key[capacity];
115         auto tempValueData = new Value [capacity];
116         for (int i = 0; i < back; i++) {
117             tempKeyData[i] = keyData[i];
118             tempValueData[i] = valueData[i];
119         }
120         delete[] keyData;
121         delete[] valueData;
122         keyData = tempKeyData;
123         valueData = tempValueData;
124     }
125
126     /* Appends new data to the end of the array. */
127     SET_DATA(key, value, back);
128     int parent = (back - 1) / 2;
129     int child = back;
130
131     /* Heapify all nodes. */
132     while (parent >= 0 && keyData[parent] < keyData[child]) {
133         SWAP(parent, child);
134         child = parent;
135         parent = (child - 1) / 2;
136     }
137     back++;
138 }
139
140 template<typename Key, typename Value>
141 void PriorityQueue<Key, Value>::pop() {
142     if (back > 0) {
143         back--;
144         keyData[0] = keyData[back];
145         valueData[0] = valueData[back];
146         int parent = 0;
147         int child = parent * 2 + 1;
148         bool placed = false;
149
150         /* Heapify all nodes. */
151         while (!placed && child < back) {
152             if (child < back - 1 && keyData[child] < keyData[child + 1])
153                 child += 1;
154             /* Heapify complete. */
155             if (keyData[parent] >= keyData[child])
156                 placed = true;
157             else {
158                 SWAP(parent, child);
159             }
160             parent = child;
161             child = parent * 2 + 1;
162         }
163     }
164 }
165
166 template<typename Key, typename Value>
167 Value PriorityQueue<Key, Value>::top() {
168     if (back != 0){
169         return valueData[0];
170     } else {
171         return nullptr;
172     }
173 }
174
175 #endif //SPA_PICO_PRIORITY_QUEUE_H
```

Index