3種類のキューの構造と実装

|  |  |
| --- | --- |
| 報告者 | XXXXXXX 〇〇　〇〇 |
| 提出日 | 20XX/XX/XX |

3種類のキューの構造と実装

提出者: 〇〇　〇〇

学籍番号: xxxxxxxx

提出日: 2023/03/29

# 目的

一般的に3種類のキューがよく知られている．それらは，再構築の仕組みを持つキュー，リングバッファの仕組みを持つキュー，リスト構造のキューである．この実験は，これらのキューの構造を理解し，実装することを目的とする．

# 原理

キューの特徴と3種類のキューの構造について説明する．

## キュー

キューとは，FIFO(First In First Out) または LILO(Last In Last Out)，つまり，先入れ先出しまたは後入れ後出しと呼ばれるデータ構造である．図 1にキューのイメージを示す．この図にある丸で囲まれた数字はデータが追加された順番を表している．キューではこの順番で追加されたデータを取り出すとき，もっとも最初に追加されたデータである①の5が取り出される．このようにデータを取り出すことをdequeueという．また，新たなにデータを追加するときには，⑤の6が図では一番左側に追加される．このようにデータを追加することをenqueueという．

キューはスタックとは異なり，キュー内にある一番古いデータから取り出されていくことがわかる．



図 1: キューの模式図

## 再構築の仕組みを持つキュー

dequeue した際にキュー内のデータが1つなくなったことをプログラムで書かないと問題が発生する．今回はこのデータを削除してキューを正常に使えるような仕組みを持つキューを再構築の仕組みを持つキューと呼ぶ．このプログラムでは，enqueueしたデータを配列の要素番号が大きいほうに収めていき，dequeueしたらそのデータを配列から削除することで上記動作を実現する．言い換えればdequeueする要素番号は0であり，そのあとの続く要素をすべて1つずつずらしていく必要がある．

## リングバッファの仕組みを持つキュー

再構築の仕組みを持つキューには問題がある．それは計算量のオーダがn(ここでnはデータの要素数)となることである．なぜならば，dequeueしたのちに要素番号0の要素が削除されるとともに，要素番号1からnまでについてはデータをすべてコピーする必要があるためである．これを解決できるのがリングバッファの仕組みを持つキューである．

リングバッファの仕組みを持つキューのイメージを図 2に示す．buffer\_sizeはキューに入れることができるデータ量，next\_point は次にデータを入れる場所，quantity はキューに入っているデータ量である．enqueue する際は next\_pointにデータを入れて next\_point，quantity を1増やす．また，dequeue する際は next\_point -quantity に buffer\_size を足したものを buffer\_size によって余剰を取ると一番古いデータの場所が得られるため，それを使いデータを取り出す．このような構造にすることで，再構築が不必要となるため，dequeue の計算量もオーダ 1 とすることができる．



図 2: リングバッファの模式図

## リスト構造のキュー

いままで説明した構造のキューでは，はじめに指定したキューのサイズ分でしかデータを格納することができない．しかし，リスト構造のキューを使用すると，コンピュータのメモリがいっぱいになるまでデータを格納することができるようになる．

リスト構造のキューのイメージを図 3に示す．bottom\_queueおよびtop\_queue はそれぞれ一番古いデータのある場所，一番新しいデータのある場所を格納するポインタである．この構造のキューは値と次のデータが格納されているアドレスで構成されている．そのため，値が格納されているメモリの場所がバラバラであっても，全てのデータをたどっていくことができる．

# 実験環境



図 3: リスト構造のキュー

3 種類のキューを実装した環境を表 1に示す．

表 1: 実験環境

|  |  |
| --- | --- |
| OS | Windows 10 Home |
| 統合開発環境(IDE) | Visual Studio 2019 |
| CPU | AMD RYZEN 5 2.38[GHz] |
| メモリ | 8.0[GiB] |
| SSD | 512[GiB] |

# プログラムの設計と説明

ここでは，3 種類のキューの違いがよく出る enqueue，dequeue，showQueue 関数の設計について説明する．なお，ここに書かれる変数名と実際にプログラムに書かれている変数名は一致している．

## 再構築の仕組みを持つキュー

再構築の仕組みを持つキューでは，要素数 buffer\_size の配列 data とデータ量を quantity と定義することでキューを構成する．

### enqueue 関数

enqueue 関数によってデータをキューに格納する手順を説明する．まず，キューに空きがあり，正の整数が格納する値として入力された場合の説明をする．キューの一番上にデータを追加することは，データ量を示す quantity を要素番号としてデータを格納することで実現できる．そしてデータ格納後，データ量が増えるため quantity をインクリメントする．これが，正常に enqueue の動作を行った場合である．次に，エラー処理について説明する．quantity が buffer\_size と等しかった場合，キューは満杯であり，データを格納することができないため，エラーコードを返す．また，負の整数がキューに格納する値として入力された場合もエラーコードを返す．この2つが想定するエラーに対する処理である．

### dequeue 関数

dequeue 関数によってキューからデータを取り出す手順を説明する．まず，キューにデータが存在し，データを取り出す場合の説明をする．再構築の仕組みを持つキューでは，取りだすデータは配列の要素番号が必ず0である．その値を取得し，quantity をデクリメントする．このまま取得したデータを返すだけでは，要素番号0のデータはすでに取り出されてしまっているため問題が発生する．そのため再構築を行う．再構築は，for 文を使用し，キュー内の古いほう，つまり要素番号1のデータから昇順に1つずつ要素を下げる，つまり注目する要素番号+1の要素をコピーしていくことで実現させる．これがデータを取りだした後に再構築を行う手順である．また，quantity が 0 である場合，キューは空であるため，これ以上データを取り出すことはできない．そのため，エラーコードを返す．

### showQueue 関数

ここでは，キューを初期化する関数が正常に動作しているかを確認するために，キューの中身全てを表示する．for 文を使用し，要素番号を0からbuffer\_size 未満の間，1ずつ増やし，その要素番号のデータを表示する．これがキューの中身を表示する手順である．

## リングバッファの仕組みを持つキュー

リングバッファの仕組みを持つキューの，キューを構成する構造体 Queue をリスト 1に示す．構造体Queue は，データが入る配列，次にデータを入れる場所の配列番号，データの個数でキューを構成される．

リスト 1: リングバッファの仕組みを持つキューの構造

|  |
| --- |
| 1. typedef struct { 2. int array[QUEUE\_SIZE]; //データが入る配列 3. int next\_point; //次にデータを入れる場所の配列番号 4. int quantity; //データの個数 5. }Queue; |

### enqueue 関数

enqueue 関数によってデータをキューに格納する手順を説明する．データを追加するときには 要素番号next\_point の要素に追加すればよい．そして，quantityとnext\_pointをインクリメントし終了する．しかし，next\_pointのインクリメントは考慮しなくてはならないことがある．図 2を見ると，要素番号3の次に0がつながっていることがわかる．つまり，next\_pointがbuffer\_sizeに等しくなったとき，0 に戻さなければいけない．以上がデータを格納する手順である．また，想定するエラーは，再構築の仕組みを持つキューと同じである．

### dequeue 関数

dequeue 関数によってキューからデータを取り出す手順を説明する．はじめに，データを取りだす場所について説明する．一番古いデータの場所は基本的に next\_point - quantity で与えられる．しかし，quantity より next\_point の方が小さく，マイナスになる場合がある．これに対応するために，next\_point - quantity に buffer\_sizeを足したものを buffer\_sizeで余剰を得る．これによって，どの場合でも一番古いデータの場所を示すことができる．図 4はデータを取り出す場所についてのイメージである．これは，円で表されるリングバッファの仕組みを持つキューを縦に無限に続き，同じ場所が繰り返されるキューとして表している．( ) 内は next\_point - quantity + buffer\_size ，その右の数字は実際の要素番号を示す．

次に，データを取り出すまでを説明する．上で説明した方法で求めた，データを取り出す場所を使い，データを取得しておく．取得したらその場所の値を0にして，データが入っていない状態にする．そして，quantityをデクリメントする．最後に取得した一番古いデータを返して終了する．これがデータを取り出す手順である．また，enqueue 関数と同じく，想定するエラーは再構築の仕組みを持つキューと同じである．



図 4: リングバッファのキューを縦に続くキューと見立てたときのイメージ

### showQueue 関数

showQueue 関数では，キューがリングバッファの仕組みになっていることを確認するため，配列番号 0 から buffer\_size 未満まで順番に表示している．これは，再構築の仕組みを持つキューの showQueue 関数と表示方法が同じである．

## リスト構造のキュー

キューの構成について説明する．リスト構造のキューの構造体 queue リスト 2に示す．リングバッファの仕組みを持つキューでは，構造体 Queue を1つ宣言すると，その構造体のみでキューをなした．一方，リスト構造のキューでは構造体 Queue を1つ宣言すると，1つのデータを入れる箱のように取り扱えるようになる．それを多く宣言し，それぞれの構造体 Queue のインスタンスをリンクさせることでキューのデータ構造となる．図 3 を使い説明する．3 つの箱があるが，この箱それぞれが構造体 Ｑueue の1つである．この構造体 Queue には値と次の箱を示すアドレスが格納されていて，そのアドレスをたどることで3つの箱が1つのキューとなっている．また，最古，最新のキューのアドレスを記憶しておくポインタを使うことにより，enqueue，dequeue，キューの表示を行える．

リスト 2: リスト構造のキューの構造

|  |
| --- |
| 1. typedef struct queue\_t 2. { 3. int val; //値 4. struct queue\_t\* addr; //次の値が格納されているqueue のアドレス 5. }Queue; 6. Queue\* bottom\_queue = NULL; //最古のキューのアドレスを記憶しておくポインタ 7. Queue\* top\_queue = NULL; //最新のキューのアドレスを記憶しておくポインタ |

リスト構造のキューでは今までの関数に加え，動的に確保したデータ領域をすべて解放する関数freeQueueについても説明する．

### enqueue 関数

enqueue 関数によってデータをキューに格納する手順を説明する．はじめに，データを格納するキューのポインタを宣言する．次に，malloc によって構造体 Queue の1個分の領域を確保し，その先頭アドレスを宣言したポインタに入れる．正常にポインタにアドレスが格納されたとき，ポインタが指し示す場所にデータを格納する．この次に，データを入れる箱のアドレスによって，全ての箱をリンクさせるが，このとき，2つの状態を想定する．1つがキューにデータが存在しないときであり，もう1つがキューに１つ以上のデータが存在するときである．まず，キューにデータが存在しないときの手順について説明する．bottom\_queue に格納されているアドレスが NULL のとき，キューにデータが存在しない．そのとき，bottom\_queue，top\_queue のどちらにも malloc によって確保した領域の先頭アドレスを格納する．

次に，データが1つ以上のデータが存在する時の手順について説明する．はじめに，今まで最新であった箱，つまり，ポインタ top\_queue が指し示している構造体 Queue の次の構造体Queueのアドレスを格納する場所に，今回mallocによって確保したメモリの先頭アドレスを格納する．そして，今回確保したメモリに格納される値が最新となるため，その先頭アドレスを top\_queue に格納する．以上がデータを格納する手順である．また，想定するエラーは入力された値が負の整数である場合とメモリが満杯のためメモリ確保ができない場合の2つである．

### dequeue 関数

dequeue関数によってキューからデータを取りだす手順を説明する．はじめに，新たに最古となるキューのアドレスを一時的に指しておくポインタを宣言する．次に，2つの場合を考える．1つはキューに1つのデータが格納されている場合で，もう1つは2つ以上のデータが格納されている場合である．まず，1つのデータが格納されている場合の手順を説明する．はじめに，最古のデータを返り値にするために取り出す．データを取り出した後，そのデータを格納していたメモリを解放する．この時点でキューにデータがない状態になる．そのため，bottom\_queue と top\_queueにNULLを入れる．これで1つのデータが格納されている場合の処理を終える．

次に，2つ以上のデータが格納されている場合の手順について説明する．はじめに，最古のデータを取りだしておく．次に，新たに最古となるキューのアドレスを一時的に指しておくポインタにbottom\_queueの指す構造体Queueのaddrに格納されているアドレスを入れる．そして，bottom\_queueの指す構造体Queueを解放し，bottom\_queueへ新たに最古となるキューのアドレスを入れる．これで2つ以上のデータが格納されている場合の処理を終える．

最後に，取りだしておいたデータを戻り値として返して終了する．これがデータを取り出す手順である．また，bottom\_queueにNULL が入っている場合，キューが空であるため，エラーコードを返す．

### showQueue 関数

showQueue関数では，bottom\_queue から順に箱のアドレスをたどっていき，NULL となったらプリントを終了する．この手順について説明する．はじめに，プリントするデータが入っている構造体Queueのアドレスを格納するポインタ宣言する．このとき，bottom\_queueのアドレスで初期化する．次に，while文を使用し，宣言したポインタが NULL でない間，以下のことを行う，まず，ポインタが指し示す構造体Queueに格納されている値を表示する．次に，ポインタが指し示す構造体Queueに格納されているアドレスを新たにポインタに入れる．これが while文で行うことである．このようにリスト構造のキューではデータの中身の表示を行う．

### freeQueue 関数

プログラムを終えるとき，mallocによって動的に確保したメモリを全て解放しなければ，管理されていないメモリが増大し，いずれはメモリが不足する可能性がる．そのため，プログラムを終える前にこの関数を呼び出すことで，その時点で確保している構造体Queue の箱を全て解放する．

キューを全て解放する手順について説明する．はじめに，解放する構造体 Queue を指し示すポインタを宣言する．次に，while文を使用し，bottom\_queue が NULLでない間，以下のことを行う．まず，宣言したポインタに bottom\_queue に入っているアドレスを格納する．次に，最古のキューを解放した後，新たに最古となるキューのアドレスを bottom\_ queue に入れる．そして，宣言したポインタが指し示す構造体Queueをfree関数により解放する．これが while 文で行うことである．このようにしてキューのメモリを全て解放する．

# プログラム

ここでは 3 種類のキューを実装するときに作成した関数の一部を示す．

## 再構築の仕組みを持つキュー

リスト 3は3ページの4.1再構築の仕組みを持つキューで説明した関数(抜粋)である．

リスト 3: 再構築の仕組みを持つキューの関数(抜粋)

|  |
| --- |
| 1. // キューにデータを格納する関数 2. // データ型 int 型 3. // 引数 4. // int 型 data : キューに格納する値 5. // 戻り値 6. // -100 : enqueue 成功 7. // -101 : キューの残領域がない 8. // -102 : 0以下のデータが入力された 9. // 10. int enqueue(int data) 11. { 12. /\* 中略 \*/ 13. return -100; 14. } 15. // キューからデータを取り出す関数 16. // データ型 int 型 17. // 引数 18. // なし 19. // 戻り値 20. // 取りだしたデータ : 取りだしに成功した場合 21. // -201 : エラーコード (取りだせるデータがない) 22. // 23. int dequeue(void) 24. { 25. int tmp; 26. int i; 27. /\* 中略 \*/ 28. return tmp; 29. } 30. // キューの配列のデータを表示する関数 31. // データ型 int 型 32. // 引数 33. // なし 34. // 戻り値 35. // 0 : 表示成功 36. // 37. int showQueue(void) 38. { 39. int i; 40. /\* 中略 \*/ 41. return 0; 42. } |

## リングバッファの仕組みを持つキュー

リスト 4は4ページの4.2リングバッファの仕組みを持つキューで説明した関数のみを表示したものである．

リスト 4: リングバッファの仕組みを持つキューの関数(抜粋)

|  |
| --- |
| 1. // キューにデータを格納する関数 2. // データ型 int 型 3. // 引数 4. // queue 型ポインタ obj : データを追加するキューの構造体のアドレス 5. // int 型 data : キューに格納するデータ 6. // 戻り値 7. // -100 : 成功 8. // -101 : キューの残領域がないとき 9. // -102 : 自然数でないデータが入力されたとき 10. // 11. int enqueue(queue\* obj, int data) 12. { 13. /\* 中略 \*/ 14. return -100; 15. } 16. // キューからデータを取り出す関数 17. // データ型 int 型 18. // 引数 19. // queue 型ポインタ obj : データを取りだすキューの構造体のアドレス 20. // 戻り値 21. // 取りだしたデータ : 取りだしに成功した場合 22. // -201 : エラーコード (取りだせるデータがない) 23. // 24. int dequeue(queue\* obj) 25. { 26. int tmp; 27. /\* 中略 \*/ 28. return tmp; 29. } 30. // キューの配列のデータを表示する関数 31. // データ型 int 型 32. // 引数 33. // なし 34. // 戻り値 35. // -500 : 表示成功 36. // 37. int showQueue(queue \*obj) 38. { 39. int i; 40. /\* 中略 \*/ 41. return -500; 42. } |

## リスト構造のキュー

リスト 5は5ページの4.3リスト構造のキューで説明した関数のみを表示したものである．

リスト 5: リスト構造のキューの関数(抜粋)

|  |
| --- |
| 1. // キューにデータを格納する関数 2. // データ型 int 型 3. // 引数 4. // int 型 data : キューに格納する値 5. // 戻り値 6. // -100 : 成功 7. // -101 : キューの残領域がない 8. // -102 : 0以下のデータが入力されたとき 9. int enqueue(int data) 10. { 11. int r\_val = -100;//返り値を宣言する 12. Queue\* new\_queue;//新しく確保する領域（構造体）へのポインタを宣言する 13. /\* 中略 \*/ 14. return r\_val; 15. } 16. // キューからデータを取り出す関数 17. // データ型 int 型 18. // 引数 19. // なし 20. // 戻り値 21. // r\_val : dequeue で取りだした値 22. // -201 : キューが空のとき 23. // -202 : キュー取り出しエラー 24. // -200 : その他のエラー 25. int dequeue(void) 26. { 27. int r\_val = -200; //返り値を入れる変数を確保する 28. Queue\* new\_bottom; //新たに最古になる領域を指すポインタを宣言する 29. /\* 中略 \*/ 30. return r\_val; 31. } 32. // キューのデータを表示する関数 33. // データ型 int 型 34. // 引数 35. // なし 36. // 戻り値 37. // 0 : 表示成功 38. int showQueue(void) 39. { 40. Queue\* look\_queue = bottom\_queue; 41. /\* 中略 \*/ 42. return 0; 43. } 44. // キューのデータ領域全てを解放する関数 45. // データ型 void 型 46. // 引数 47. // なし 48. // 戻り値 49. // なし 50. // 51. void freeQueue(void) 52. { 53. Queue\* this\_queue; 54. /\* 中略 \*/ 55. } |

# 実行結果

この章では実装した 3 種類のキューについて，関数が正常に動作しているかを確認できるように表示した結果を示す．

## 再構築の仕組みを持つキュー

以下に再構築の仕組みを持つキューの実行結果を示す．実行結果では，次の確認をしている．

* 正常に enqueue されている．
* 正常に dequeue されている．
* dequeue した際に再構築が行われている．
* エラーコードに対して，正しいエラー内容が表示されている．

上記のそれぞれそれについて説明を行う．まず，正常に enqueue されているかを確認する．実行結果の「キューが満杯になるまでデータを格納する」というところより，queue に順に値が入力されているため，正常に動作していることがわかる．次に，正常に dequeue されているかを確認する．「キューがカラになるまでdequeue する」というところより，値が enqueue された順番で queue から取りだしているため，正常に動作していることがわかる．また，データを取り出したときに，残ったデータが左詰めのようにキューの中で再構築されていることも確認できる．最後に，エラーコードに対して正しいエラー内容が表示されていることを確認する．「負の値 (-1) を enqueue する」，「キューが満杯の状態で 1000 を enqueue する」，「キューが空の状態で dequeue する」のところからわかるように，内容に応じて，正しいエラー内容が表示されていることが確認できる．

キューの大きさ:4

データ格納前のキュー

queue :| 0 | 0 | 0 | 0 |

負の値 (-1) を enqueue する

負の値が入力されました (-102)

queue :| 0 | 0 | 0 | 0 |

キューが満杯になるまでデータを格納する

queue :| 1 | 0 | 0 | 0 |

queue :| 1 | 2 | 0 | 0 |

queue :| 1 | 2 | 3 | 0 |

queue :| 1 | 2 | 3 | 4 |

キューが満杯の状態で 1000 を enqueue する

キューが満杯でデータを格納できません (-101)

queue :| 1 | 2 | 3 | 4 |

キューがカラになるまで dequeue する

取りだしたデータ: 1

queue :| 2 | 3 | 4 | 0 |

取りだしたデータ: 2

queue :| 3 | 4 | 0 | 0 |

取りだしたデータ: 3

queue :| 4 | 0 | 0 | 0 |

取りだしたデータ: 4

queue :| 0 | 0 | 0 | 0 |

キューが空の状態で dequeue する

キューが空でデータを取り出せません (-201)

queue :| 0 | 0 | 0 | 0 |

## リングバッファの仕組みを持つキュー

以下にリングバッファの仕組みを持つキューの実行結果を示す．実行結果では，次の確認をしている．

* 正常に enqueue されている．
* 正常に dequeue されている．
* エラーコードに対して，正しいエラー内容が表示されている．
* リングバッファの仕組みになっている．

上記のそれぞれそれについて説明を行う．正常に enqueue されていること，正常に dequeue されていること，エラーコードに対して，正しいエラー内容が表示されていることは，再構築の仕組みを持つキューと同じように正常に動作していることが確認できる．リングバッファの仕組みになっていることを確認する．「リングバッファになっているか確認する」というところでは，enqueue，dequeue をそれぞれ 3 回ずつ行うことを 5 回繰り返すことで，リングバッファの仕組みになっていることを確認できるようにしている．ここより，配列番号がbuffer\_size と 0 をまたぐようにキューを構成していても，またその他の場合でも正常に enqueue，dequeue を行えているため，リングバッファの仕組みになっていることが確認できる．

キューの大きさ:5

データ格納前のキュー

queue :| 0 | 0 | 0 | 0 | 0 |

負の値 (-1) を enqueue する

負の値が入力されました (-102)

queue :| 0 | 0 | 0 | 0 | 0 |

キューが満杯になるまでデータを格納する

queue :| 1 | 0 | 0 | 0 | 0 |

queue :| 1 | 2 | 0 | 0 | 0 |

queue :| 1 | 2 | 3 | 0 | 0 |

queue :| 1 | 2 | 3 | 4 | 0 |

queue :| 1 | 2 | 3 | 4 | 5 |

キューが満杯の状態で 1000 を enqueue する

キューが満杯でデータを格納できません (-101)

queue :| 1 | 2 | 3 | 4 | 5 |

キューがカラになるまで dequeue する

取りだしたデータ: 1

queue :| 0 | 2 | 3 | 4 | 5 |

取りだしたデータ: 2

queue :| 0 | 0 | 3 | 4 | 5 |

取りだしたデータ: 3

queue :| 0 | 0 | 0 | 4 | 5 |

取りだしたデータ: 4

queue :| 0 | 0 | 0 | 0 | 5 |

取りだしたデータ: 5

queue :| 0 | 0 | 0 | 0 | 0 |

キューが空の状態で dequeue する

キューが空でデータを取り出せません (-201)

queue :| 0 | 0 | 0 | 0 | 0 |

リングバッファになっているか確認する

queue :| 1 | 0 | 0 | 0 | 0 |

17

queue :| 1 | 2 | 0 | 0 | 0 |

queue :| 1 | 2 | 3 | 0 | 0 |

取りだしたデータ: 1

queue :| 0 | 2 | 3 | 0 | 0 |

取りだしたデータ: 2

queue :| 0 | 0 | 3 | 0 | 0 |

取りだしたデータ: 3

queue :| 0 | 0 | 0 | 0 | 0 |

queue :| 0 | 0 | 0 | 4 | 0 |

queue :| 0 | 0 | 0 | 4 | 5 |

queue :| 6 | 0 | 0 | 4 | 5 |

取りだしたデータ: 4

queue :| 6 | 0 | 0 | 0 | 5 |

取りだしたデータ: 5

queue :| 6 | 0 | 0 | 0 | 0 |

取りだしたデータ: 6

queue :| 0 | 0 | 0 | 0 | 0 |

queue :| 0 | 7 | 0 | 0 | 0 |

queue :| 0 | 7 | 8 | 0 | 0 |

queue :| 0 | 7 | 8 | 9 | 0 |

取りだしたデータ: 7

queue :| 0 | 0 | 8 | 9 | 0 |

取りだしたデータ: 8

queue :| 0 | 0 | 0 | 9 | 0 |

18

取りだしたデータ: 9

queue :| 0 | 0 | 0 | 0 | 0 |

queue :| 0 | 0 | 0 | 0 | 10 |

queue :| 11 | 0 | 0 | 0 | 10 |

queue :| 11 | 12 | 0 | 0 | 10 |

取りだしたデータ: 10

queue :| 11 | 12 | 0 | 0 | 0 |

取りだしたデータ: 11

queue :| 0 | 12 | 0 | 0 | 0 |

取りだしたデータ: 12

queue :| 0 | 0 | 0 | 0 | 0 |

queue :| 0 | 0 | 13 | 0 | 0 |

queue :| 0 | 0 | 13 | 14 | 0 |

queue :| 0 | 0 | 13 | 14 | 15 |

取りだしたデータ: 13

queue :| 0 | 0 | 0 | 14 | 15 |

取りだしたデータ: 14

queue :| 0 | 0 | 0 | 0 | 15 |

取りだしたデータ: 15

queue :| 0 | 0 | 0 | 0 | 0 |

## リスト構造のキュー

以下にリスト構造のキューの実行結果を示す．実行結果では，次の確認をしている．

* 正常に enqueue されている．
* 正常に dequeue されている．
* エラーコードに対して，正しいエラー内容が表示されている．
* キューの中身が一回なくなってもまた enqueue することで，キューを構築される．

上記のそれぞれそれについて説明を行う．まず，正常に enqueue されているかを確認する．実行結果の「格納成功」というところより，順番に値が積まれているため，正常に動作している．次に，正常に dequeue されているかをかくにんする．「取出成功」というところより，格納された順番に値を取り出しているため，正常に動作している．次に，「0 以下の値が入力されました (-102)」，「キューが空のためデータを取りだせません (-201)」というところより，エラーコードに対して，正しいエラー内容が表示されていることが確認できる．また，メモリ確保失敗によるエラー表示は，キューに膨大な量のデータを格納しなければ表示されないため省略する．最後に，キューの中身をなくした後，enqueue することで再びキューを構築されることを確認する．-201 のエラー処理の後より，キューの中身をなくした後，enqueue することで再びキューを構築し，dequeue も行っていることを確認できる．

| 1 |<-1: 格納成功

| 1 | 2 |<-2: 格納成功

| 1 | 2 | 3 |<-3: 格納成功

| 1 | 2 | 3 | 4 |<-4: 格納成功

| 1 | 2 | 3 | 4 | 5 |<-5: 格納成功

| 1 | 2 | 3 | 4 | 5 |<-0:0 以下の値が入力されました (-102)

| 2 | 3 | 4 | 5 |->1: 取出成功

| 3 | 4 | 5 |->2: 取出成功

| 4 | 5 |->3: 取出成功

| 5 |->4: 取出成功

|->5: 取出成功

|->-201: キューが空のためデータを取りだせません (-201)

| 10 |<-10: 格納成功

| 10 | 20 |<-20: 格納成功

| 20 |->10: 取出成功

# 考察

## 目的の達成

実行結果より，再構築の仕組みを持つキュー，リングバッファの仕組みを持つキュー，リスト構造のキューそれぞれで，正常に関数が動作しているかを確認できた．そのため，3 種類のキューを実装できたといえる．また，今回の実験では，再構築の仕組みを持つキュー，リングバッファの仕組みを持つキューの2つは自分でどのような動作をするかを考え，理解しながら作成した．しかし，リスト構造のキューは自分では理解することができず，与えられたテンプレートをもとに作成した．だが，作成したことにより，最終的にリスト構造のキューを理解することができた．よって，キューの構造を理解し，実装することができたため，目的を達成できたといえる．

## 3 種類のキューの特徴と有効的な使用法

ここでは，それぞれのキューについて，その特徴と特徴を生かした活用法について考察する．

再構築の仕組みを持つキューは，配列の 0番目を一番古いデータとし，そこから順にデータが積まれ，dequeue してもその構造が変わらないという特徴がある．これは，図 1のキューのイメージと全く同じであり，今回実装した 3つのキューの中で最も理解しやすく，実装しやすいと思われる．そのため，キューの理解のために再構築の仕組みを持つキューを実装することがよい活用法だと考える．

リングバッファの仕組みを持つキューは，dequeue してもデータの場所はかわらず，配列番号の buffer\_size - 1 番目と 0 番目がつながっているような構造をしているという特徴がある．これは，再構築の仕組みを持つキューの dequeue の計算量を改善したものだと考えることができる．今回実装したリングバッファの仕組みを持つキューでは，buffer\_size が決められているため，おおよそどれくらいのデータがキューに格納されるのかわかるときに有効的に活用できると思われる．

リスト構造のキューは，データを入れる箱がメモリ内でバラバラに作られているが，アドレスをリンクさせることで1つのキューとなっているという特徴がある．このキューでは，メモリがいっぱいになるまでデータを入れることができ，また，キューのサイズは決まっておらず，データが格納されるたびに新たな箱をつくり，アドレスをリンクさせる．そのため，事前にどれくらいのデータがキューに格納されるのかが予測できないときに有効的に活用できると思われる．

# 付録: 3 種類のキューのプログラム

リスト 6，リスト 7，リスト 8はそれぞれ再構築の仕組みを持つキュー，リングバッファの仕組みを持つキュー，リスト構造のキューを実装したものである．本章で説明したプログラムに比べ，マジックナンバをなくしていること，加えて関数や定数に関するコメントについてはdoxygenで説明文が生成される形式に修正してある．

リスト 6: 再構築の仕組みを持つキュー

|  |
| --- |
| 1. #define \_CRT\_SECURE\_NO\_WARNINGS 2. #include <stdio.h> 3. /\*\* 成功 \*/ 4. #define SUCCESSFUL (0) 5. /\*\* 失敗 \*/ 6. #define FAILURE (-1) 7. /\*\* 必要なし \*/ 8. #define NO\_NEED (1) 9. /\*\* キューの残領域がない \*/ 10. #define NO\_QUEUE\_SIZE (-101) 11. /\*\* 自然数以外のデータを入力しようとした \*/ 12. #define NO\_NATURAL\_NUMBER (-102) 13. /\*\* キューから取り出せるデータがない \*/ 14. #define NO\_LEFT\_DATA (-201) 15. /\*\* キューのサイズ \*/ 16. #define QUEUE\_SIZE (4) 17. /\*\* キュー \*/ 18. int queue[QUEUE\_SIZE]; 19. /\*\* キューに入っている要素数 \*/ 20. int quantity = 0; 21. /\*\* キューにデータを格納する関数 22. \* @param[in] data データ型 int 入力されるデータ 23. \* @retval SUCCESSFUL enqueueに成功した 24. \* @retval NO\_QUEUE\_SIZE キューの残領域がない 25. \* @retval 自然数でないデータが入力された 26. \*/ 27. int enqueue(int data) 28. { 29. /\* 中略 \*/ 30. return SUCCESSFUL; 31. } 32. /\*\* キューからデータを取り出す関数 33. \* @return 取り出しに成功した場合には取り出したデータ．失敗したときにはエラーコードを返す 34. \*/ 35. int dequeue(void) 36. { 37. int tmp; 38. int i; 39. /\* 中略 \*/ 40. return tmp; 41. } 42. /\*\* キュー用の配列を 0で初期化する 43. \* @retval SUCCESSFUL 初期化成功 44. \*/ 45. int initQueue(void) 46. { 47. int i; 48. /\* 中略 \*/ 49. return SUCCESSFUL; 50. } 51. /\*\* キューの配列のデータを表示する関数 52. \* @retval SUCCESSFUL 表示成功 53. \*/ 54. int showQueue(void) 55. { 56. int i; 57. /\* 中略 \*/ 58. return SUCCESSFUL; 59. } 60. /\*\* エラーコードに合わせてエラーメッセージを表示する関数 61. \* @param[in] result エラーコード 62. \* @retval SUCCESSFUL エラーメッセージを表示成功 63. \* @retval FAILURE エラーメッセージを表示失敗 64. \* @retval NO\_NEED エラーコードを表示する必要がないので表示もしない 65. \*/ 66. int showResult(int result) 67. { 68. /\* 中略 \*/ 69. return SUCCESSFUL; 70. } 71. int main(void) 72. { 73. int result; 74. int i; 75. printf("キューの大きさ:%d\n\n", QUEUE\_SIZE); 76. initQueue(); 77. printf("データ格納前のキュー\n"); 78. showQueue(); 79. printf("負の値 (-1)をenqueue する\n"); 80. result = enqueue(-1); 81. showResult(result); 82. showQueue(); 83. printf("キューが満杯になるまでデータを格納する\n"); 84. for (i = 0; i < QUEUE\_SIZE; i++) { 85. result = enqueue(i + 1); 86. showResult(result); 87. showQueue(); 88. } 89. printf("キューが満杯の状態で 1000をenqueue する\n"); 90. result = enqueue(1000); 91. showResult(result); 92. showQueue(); 93. printf("キューがカラになるまでdequeue する\n"); 94. for (i = 0; i < QUEUE\_SIZE; i++) { 95. result = dequeue(); 96. showResult(result); 97. printf("取りだしたデータ: %d\n", result); 98. showQueue(); 99. } 100. printf("キューが空の状態でdequeue する\n"); 101. result = dequeue(); 102. showResult(result); 103. showQueue(); 104. return SUCCESSFUL; 105. } |

リスト 7: リングバッファの仕組みを持つキュー

|  |
| --- |
| 1. #define \_CRT\_SECURE\_NO\_WARNINGS 2. #include <stdio.h> 3. /\*\* 成功 \*/ 4. #define SUCCESSFUL (0) 5. /\*\* 失敗 \*/ 6. #define FAILURE (-1) 7. /\*\* 必要なし \*/ 8. #define NO\_NEED (1) 9. /\*\* キューの残領域がない \*/ 10. #define NO\_QUEUE\_SIZE (-101) 11. /\*\* 自然数以外のデータを入力しようとした \*/ 12. #define NO\_NATURAL\_NUMBER (-102) 13. /\*\* キューから取り出せるデータがない \*/ 14. #define NO\_LEFT\_DATA (-201) 15. /\*\* キューのサイズ \*/ 16. #define QUEUE\_SIZE (5) 17. /\*\* キューを管理するための構造体 \*/ 18. typedef struct { 19. /\*\* データが入る配列 \*/ 20. int array[QUEUE\_SIZE]; 21. /\*\* 次にデータを入れる場所の配列番号 \*/ 22. int wp; 23. /\*\* データの個数 \*/ 24. int quantity; 25. }Queue; 26. /\*\* キューにデータを格納する関数 27. \* @param [in,out] obj データを追加するキューの構造体のアドレス 28. \* @param [in] data キューに格納するデータ 29. \* @retval SUCCESSFUL 成功 30. \* @retval NO\_QUEUE\_SIZE キューの残領域がないとき 31. \* @retval NO\_NATURAL\_NUMBER 自然数でないデータが入力されたとき 32. \*/ 33. int enqueue(Queue\* obj, int data) 34. { 35. /\* 中略 \*/ 36. return SUCCESSFUL; 37. } 38. /\*\* キューからデータを取り出す関数 39. \* @param [in,out] obj キューからデータを取り出す関数 40. \* @return 取り出しに成功した場合には取り出したデータ，失敗した時にはエラーコード 41. \*/ 42. int dequeue(Queue\* obj) 43. { 44. int tmp; 45. /\* 中略 \*/ 46. return tmp; 47. } 48. /\*\* キューの配列を初期化する関数 49. \* @param [in,out] 初期化するキューの構造体のアドレス 50. \* @return 成功 51. \*/ 52. int initQueue(Queue\* obj) 53. { 54. int i; 55. /\* 中略 \*/ 56. return SUCCESSFUL; 57. } 58. /\*\* キューの配列のデータを表示する関数 59. \* @param [in,out] 表示するキューの構造体のアドレス 60. \* @return 成功 61. \*/ 62. int showQueue(Queue\* obj) 63. { 64. int i; 65. /\* 中略 \*/ 66. return SUCCESSFUL; 67. } 68. /\*\* エラーコードに合わせてエラーメッセージを表示する関数 69. \* @param[in] result エラーコード 70. \* @retval SUCCESSFUL エラーメッセージを表示成功 71. \* @retval FAILURE エラーメッセージを表示失敗 72. \* @retval NO\_NEED エラーコードを表示する必要がないので表示もしない 73. \*/ 74. int showResult(int result) 75. { 76. /\* 中略 \*/ 77. return SUCCESSFUL; 78. } 79. int main(void) 80. { 81. int result; 82. int i, j; 83. Queue q; 84. printf("キューの大きさ:%d\n\n", QUEUE\_SIZE); 85. initQueue(&q); 86. printf("データ格納前のキュー\n"); 87. showQueue(&q); 88. printf("負の値 (-1)をenqueue する\n"); 89. result = enqueue(&q, -1); 90. showResult(result); 91. showQueue(&q); 92. printf("キューが満杯になるまでデータを格納する\n"); 93. for (i = 0; i < QUEUE\_SIZE; i++) { 94. result = enqueue(&q, i + 1); 95. showResult(result); 96. showQueue(&q); 97. } 98. printf("キューが満杯の状態で 1000をenqueue する\n"); 99. result = enqueue(&q, 1000); 100. showResult(result); 101. showQueue(&q); 102. printf("キューがカラになるまでdequeue する\n"); 103. for (i = 0; i < QUEUE\_SIZE; i++) { 104. result = dequeue(&q); 105. showResult(result); 106. printf("取りだしたデータ: %d\n", result); 107. showQueue(&q); 108. } 109. printf("キューが空の状態でdequeue する\n"); 110. result = dequeue(&q); 111. showResult(result); 112. showQueue(&q); 113. printf("リングバッファになっているか確認する\n"); 114. for (j = 0; j < QUEUE\_SIZE; j++) { 115. for (i = 0; i < 3; i++) { 116. result = enqueue(&q, i + 1 + j \* 3); 117. showResult(result); 118. showQueue(&q); 119. } 120. for (i = 0; i < 3; i++) { 121. result = dequeue(&q); 122. showResult(result); 123. printf("取りだしたデータ: %d\n", result); 124. showQueue(&q); 125. } 126. } 127. return SUCCESSFUL; 128. } |

リスト 8: リスト構造のキュー

|  |
| --- |
| 1. #define \_CRT\_SECURE\_NO\_WARNINGS 2. #include <stdio.h> 3. #include <stdlib.h> 4. /\*\* 成功 \*/ 5. #define SUCCESSFUL (0) 6. /\*\* 失敗 \*/ 7. #define FAILURE (-1) 8. /\*\* 必要なし \*/ 9. #define NO\_NEED (1) 10. /\*\* キューの残領域がない \*/ 11. #define NO\_QUEUE\_SIZE (-101) 12. /\*\* 自然数以外のデータを入力しようとした \*/ 13. #define NO\_NATURAL\_NUMBER (-102) 14. /\*\* キューから取り出せるデータがない \*/ 15. #define NO\_LEFT\_DATA (-201) 16. /\*\* デキュー時のエラー \*/ 17. #define ERROR\_AT\_DEQUEUE (-202) 18. /\*\* リスト構造のキュー \*/ 19. typedef struct queue\_t 20. { 21. /\*\* 値 \*/ 22. int val; 23. /\*\* 次の値が格納されているqueue のアドレス \*/ 24. struct queue\_t\* addr; 25. } Queue; 26. /\*\* 最古のキューのアドレスを記憶しておくポインタ \*/ 27. Queue\* bottom\_queue = NULL; 28. /\*\* 最新のキューのアドレスを記憶しておくポインタ \*/ 29. Queue\* top\_queue = NULL; 30. /\*\* キューにデータを格納する関数 31. \* @param [in] data キューに格納する値 32. \* @retval SUCCESSFUL 成功 33. \* @retval NO\_QUEUE\_SIZE メモリが足りない 34. \* @retval NO\_NATURAL\_NUMBER 自然数以外のデータを入力しようとした 35. \*/ 36. int enqueue(int data) 37. { 38. int r\_val = SUCCESSFUL;//返り値を宣言する 39. Queue\* new\_queue;//新しく確保する領域（構造体）へのポインタを宣言する 40. /\* 中略 \*/ 41. return r\_val; 42. } 43. /\*\* キューからデータを取り出す関数 44. \* @retval r\_val 取り出したデータ 45. \* @retval NO\_LEFT\_DATA キューから取り出せるデータがない 46. \* @retval ERROR\_AT\_DEQUEUE デキュー時のエラー 47. \*/ 48. int dequeue(void) 49. { 50. int r\_val = ERROR\_AT\_DEQUEUE; //返り値を入れる変数を確保する 51. Queue\* new\_bottom; //新たに最古になる領域を指すポインタを宣言する 52. /\* 中略 \*/ 53. return r\_val; 54. } 55. /\*\* キューのデータを表示する関数 56. \* @return SUCCESSFUL 表示成功 57. \*/ 58. int showQueue(void) 59. { 60. Queue\* look\_queue = bottom\_queue; 61. /\* 中略 \*/ 62. return SUCCESSFUL; 63. } 64. /\*\* エラーメッセージを表示する関数 65. \* @param [in] result エラーコード 66. \* @retval SUCCESSFUL エラーメッセージ表示成功 67. \* @retval NO\_NEED エラーコードを表示する必要がないので表示もしない 68. \*/ 69. int showResult(int result) 70. { 71. /\* 中略 \*/ 72. return SUCCESSFUL; 73. } 74. /\*\* キューのデータ領域全てを解放する関数 \*/ 75. void freeQueue(void) 76. { 77. Queue\* this\_queue; 78. /\* 中略 \*/ 79. } 80. int main(void) 81. { 82. int result; 83. result = enqueue(1); showQueue(); printf("<-1"); showResult(result); 84. result = enqueue(2); showQueue(); printf("<-2"); showResult(result); 85. result = enqueue(3); showQueue(); printf("<-3"); showResult(result); 86. result = enqueue(4); showQueue(); printf("<-4"); showResult(result); 87. result = enqueue(5); showQueue(); printf("<-5"); showResult(result); 88. result = enqueue(0); showQueue(); printf("<-0"); showResult(result); 89. result = dequeue(); showQueue(); printf("->%d", result); showResult(result); 90. result = dequeue(); showQueue(); printf("->%d", result); showResult(result); 91. result = dequeue(); showQueue(); printf("->%d", result); showResult(result); 92. result = dequeue(); showQueue(); printf("->%d", result); showResult(result); 93. result = dequeue(); showQueue(); printf("->%d", result); showResult(result); 94. result = dequeue(); showQueue(); printf("->%d", result); showResult(result); 95. result = enqueue(10); showQueue(); printf("<-10"); showResult(result); 96. result = enqueue(20); showQueue(); printf("<-20"); showResult(result); 97. result = dequeue(); showQueue(); printf("->%d", result); showResult(result); 98. freeQueue(); 99. return SUCCESSFUL; 100. } |