גירסה 1.00 - 28.6.2003

רשימת דילוגים

מסמך זה הורד מהאתר http://underwar.livedns.co.il.
אין להפיץ מסמך זה במדיה כלשהי, ללא אישור מפורש מאת המחבר.
מחבר המסמך איננו אחראי לכל נזק, ישיר או עקיף, שיגרם עקב השימוש במידע המופיע במסמך, וכן לנכונות התוכן של הנושאים המופיעים במסמך. עם זאת, המחבר עשה את מירב המאמצים כדי לספק את המידע המדויק והמלא ביותר.

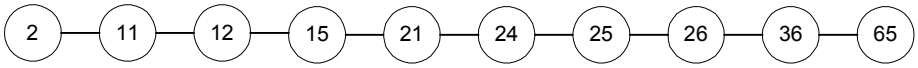
Nir Adar
Email: underwar@hotmail.com
Home Page: http://underwar.livedns.co.il

אנא שלחו תיקונים והערות אל המחבר.

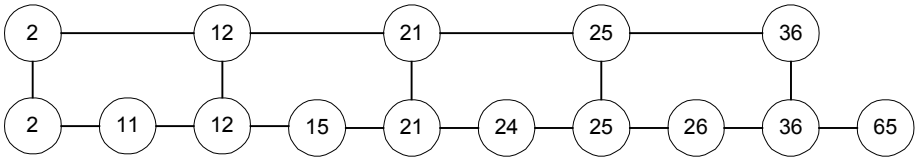
רשימת דילוגים

עבור עצים מאוזנים, מימוש פעולות המילון נעשה בזמן $O(\log n)$ במקרה הגרוע ביותר.
מימוש הפעולות אינו טריוויאלי, במיוחד כאשר יש צורך לתמוך גם בפעולות הדורשות לשמור בכל צומת אינפורמציה ייחודית (כמו rank) ולעדכן אותה.
נציג מבנה נתונים בעל מימוש פשוט יותר, שלא דורש איזונים לאחר הכנסה/הוצאה.
בתמורה לכך, ביצוע פעולות המילון יעשה בזמן $O(\log n)$ בממוצע.
ראינו כבר שעצי חיפוש ללא פעולות איזון יכולים לשמש למימוש פעולות המילון בזמן $O(\log n)$ בממוצע. בניתוח שעשינו הנחנו שהוכנסו n איברים לעץ, לפיכך קיימות !n פרמוטציות להכנסתם. הממוצע של גובה !n העצים שנוצרו הוא $O(\log n)$.
בניתוח זה ישנן מספר בעיות. ראשית, בדרך כלל ההסתברות להופעת פרמוטציה בקלט איננה אחידה. שנית, מבנה הנתונים שנוצר תלוי בסדר הכנסת הנתונים. אדם/תהליך המחבל במערכת יכול להכניס סדרת נתונים כך שהפעולות יבוצעו בזמן האיטי ביותר.
במבנה שנציג - רשימת דילוגים, בעיות אלו נפתרות. הפעולות וזמן ביצוען תלוי בהגרלות שעושה המחשבים, ולא בסדר הכנסת הנתונים. הממוצע מחושב לפי ההגרלות ולא לפי הנחות על ההסתברות של הקלטים. אלגוריתם כזה נקרא **אלגוריתם רנדומלי**.

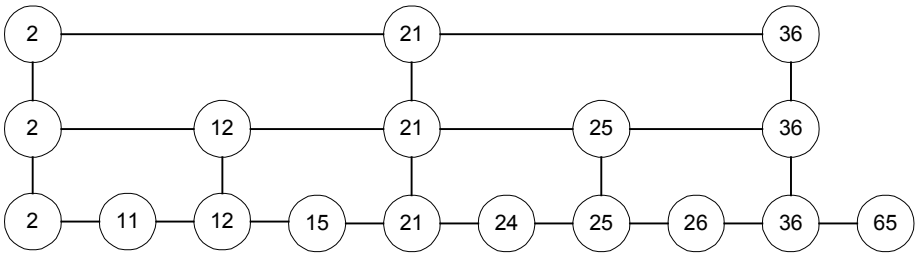
נביט ברעיון של מבנה הנתונים.
ניקח רשימה ממוינת של איברים. חיפוש איבר בסוף הרשימה הוא יקר (מבחינת פעולות).



כדי לזרז את החיפוש (פי 2), נוסיף מדריך - תת רשימה של כל איבר שני. ממדריך זה יהיה אפשר להגיע לאיבר המתאים ברשימה הראשונה.

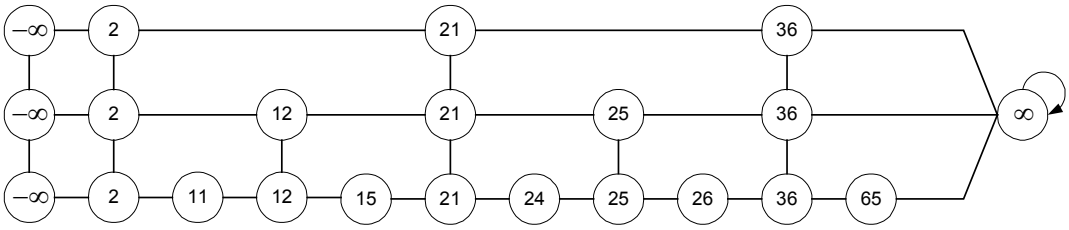


כדי לזרז את החיפוש במדריך נוסיף רמה נוספת:



כך נמשיך עד לרמה $\lceil \log n \rceil$ בה יהיה איבר בודד.

כדי לפשט את האלגוריתמים המממשים רשימת דילוגים, נהוג להוסיף צומת דמה בתחילת כל שורה ברשימת הדילוגים, וכן צומת אחת בסוף רשימת הדילוגים, שכל השורות יסתיימו בה.



<u>אורך מסלול החיפוש</u>

במהלך החיפוש, במעבר מרמה לרמה אנו עוברים על פני שני מצביעים לכל היותר. לפיכך, אורך המסלול הוא $2\log n$. מספר הצמתים הכולל אינו עולה על $2n$.

<u>פעולת חיפוש ברשימת דילוגים (id)</u>

1. p מקבל את כתובתו של הצומת העליון השמאלי ביותר.
2. כל עוד לא הגעת אל תחתית רשימת הדילוגים או לא הגעת אל סופה, בצע:
3. כל עוד $p.right.id \le id$ בצע: $p \leftarrow p.right$.
4. אם לא הגעת אל תחתית הרשימה, בצע $p \leftarrow p.down$.
5. אם $p.id = id$ החזר את p. אחרת החזר: "האיבר המבוקש איננו ברשימת הדילוגים".

<u>פעולת ההכנסה ברשימת דילוגים</u>

1. חפש את k. אם k נמצא, סיים.
2. שמור מצביע לצומת הימני ביותר בכל רמה במסלול החיפוש.
3. הוסף צומת חדש ברמה התחתונה ביותר וקבע את המפתח להיות k.
4. לפי סדר הרמות מלמטה למעלה:
   הגרל מטבע (toss).
   אם יוצא 0: הוסף צומת חדש מעל הרמה הנוכחית וקבע את המפתח בו להיות k.
   אם יוצא 1: עצור.
5. אם ברמה העליונה הוגרל 0, הוסף רמה חדשה.

<u>פעולת הוצאה מרשימת דילוגים</u>

1. מצא את האיבר בעל המפתח k.
2. הוצא איבר זה מכל הרמות בהן הוא מופיע.

<u>מספר הצמתים ברשימת דילוגים</u>

מספר הצמתים הממוצע ברשימת דילוגים הוא 2n, כאשר n הוא מספר המפתחות במבנה.

<u>אורך מסלול החיפוש הממוצע</u>

האורך הממוצע L של מסלול חיפוש ברשימת דילוגים עם n מפתחות מקיים $L \le 2\log_2 n + 2$.

<u>זמן ביצוע הפעולות</u>

חיפוש, הכנסה, והוצאה מרשימת דילוגים נעשים בזמן ממוצע $O(\log n)$ כיוון שבכל צעד על מסלול החיפוש מתבצעים $O(1)$ צעדים.

רשימת דילוגים דטרמיניסטית

כאשר אנו משתמשים ברשימת הדילוגים הרנדומלית, הפעולות וזמן ביצוען תלוי בהגרלות שעושה המחשבים, ולא בסדר הכנסת הנתונים. בממוצע אנו מקבלים זמני ביצוע פעולות של $O(\log n)$.
אולם, רשימת הדילוגים הרנדומלית מתבססת על כך שמנוע יצירת המספרים האקראיים של המחשב מתנהג "כמצופה". במידה והמצב הוא לא כזה, עשויה רשימת הדילוגים להראות כרשימה מקושרת פשוטה, וסיבוכיות הפעולות עליה יהיו ליניאריות במקרה הגרוע בהתאם.
נרצה למצוא דרך להשתמש ברשימת דילוגים, בעזרתה נוכל להגיע לזמני ביצוע פעולות של $O(\log n)$ במקרה הגרוע.
כידוע - קיימים מבני נתונים, עצי חיפוש, המאפשרים לממש מילון ב- $O(\log n)$ - עצי AVL, עצי 2-3, עצי red-black ועוד. אנו מחפשים אלטרנטיבה למבנים אלו עקב המימוש המסובך שלהם.
נציג פתרון המשתמש ברשימת דילוגים בהדרגתיות, תוך כדי הצגת הדרך כיצד פותח הפיתרון הסופי בו נשתמש.

<u>נקודת ההתחלה</u>

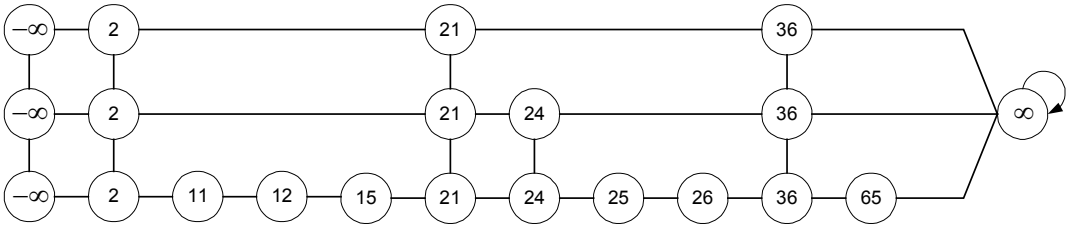
<u>הגדרה</u>

רשימת דילוגים L תכונה **רשימת דילוגים מאוזנת בשלמות** אם היא מקיימת את התנאים הבאים:
1. L היא רשימת דילוגים.
2. עבור k טבעי קבוע כלשהו, מתקיים כי כל איבר k-י בגובה h ברשימת הדילוגים מופיע גם בגובה h+1 ברשימת הדילוגים.

עבור רשימת דילוגים מאוזנת בשלמות, פעולת חיפוש מתבצעת בזמן לוגריתמי.
אם זאת, איננו יכולים לבצע פעולות הכנסה או הוצאה על הרשימה, מבלי לפגוע בהיותה רשימת דילוגים מאוזנת בשלמות.
לפיכך, נחליש מעט את הדרישה השניה, על ידי שינוי הביטוי "עבור <u>כל</u> איבר k-י".

<u>הגדרה</u>

תהי רשימת דילוגים L בעלת n מפתחות.
נאמר ששני אלמנטים ברשימה **מקושרים** אם קיים מצביע אחד או יותר ביניהם.
בהינתן שני אלמנטים מקושרים, אחד בגובה h בדיוק ($h>1$), והשני בגובה h או יותר, נגדיר את **המרווח** ביניהם להיות מספר האלמנטים ברמה ה-$h-1$ הקיימים ביניהם.
לדוגמא, נביט ברשימת הדילוגים הבאה:



המרווח בין 21 ל-36 הוא 1, המרווח בין 24 ל-36 הוא 2, ואילו המרווח בין 2 ל-21 הוא 0.

הגדרה

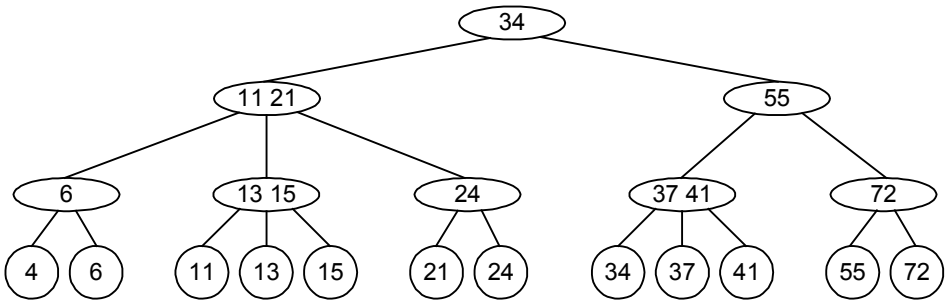
רשימת דילוגים L תיקרא **רשימת דילוגים דטרמינסטית 1-2** אם מתקיים:
1. L היא רשימת דילוגים.
2. המרווח בין כל שני אלמנטים ב-L הוא 1 או 2.

טענה

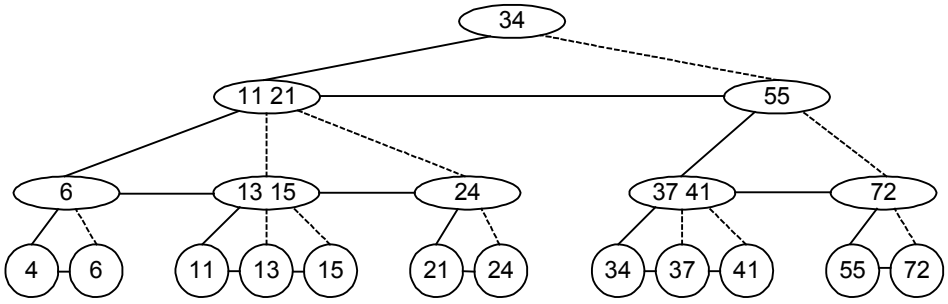
בין רשימת דילוגים לעצי 2-3 יש קשר הדוק. קיימת התאמה חד חד ערכית ועל בין כל עץ 2-3 לרשימת דילוגים דטרמיניסטית 1-2.

נציג את האלגוריתם המשמש להפיכת עץ 2-3 לרשימת דילוגים דטרמינסטית 1-2 על ידי דוגמא.

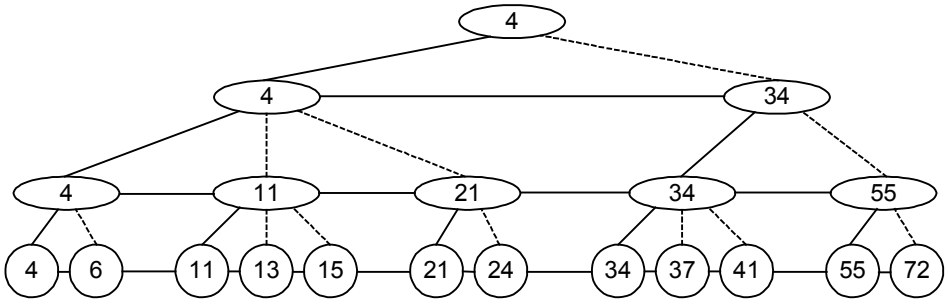
ניקח את עץ 2-3 הבא:



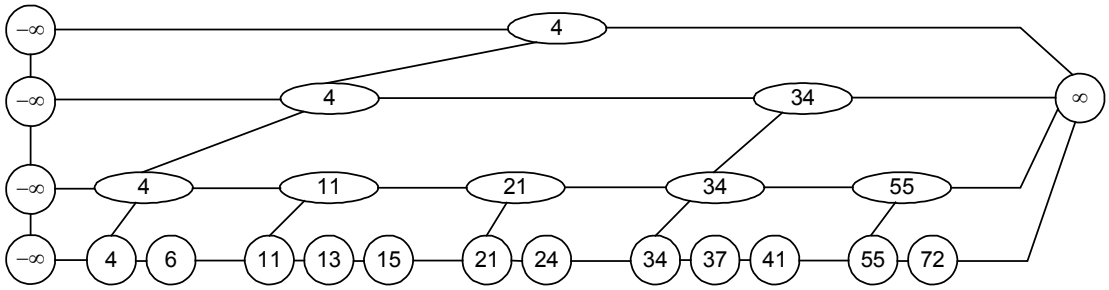
מימוש של העץ כעץ בינרי ייראה כך:



חיבור הצמתים בכל רמה ושינוי ערכי הצמתים הפנימיים נותן skip-list:



נוכל להוסיף גם מפתחות התחלה וסיום, ולקבל את רשימת הדילוגים האקוויולנטית הבאה:



<u>מימוש פעולות מילון על רשימת הדילוגים</u>

חיפוש - חיפוש מפתח ברשימת דילוגים דטרמינסטית נעשה באופן זהה לחיפוש מפתח ברשימת דילוגים רנדומלית.
הכנסה - הכנסה נעשית על ידי חיפוש המפתח המבוקש, והכנסת המפתח ברמה התחתונה של רשימת הדילוגים, בסוף מסלול החיפוש. בשלב זה ייתכן מצב בו יהיו שלושה אלמנטים בגובה 1 הנמצאים ברצף. נפתור בעיה זו על ידי העלאת האמצעי ביניהם לרמה 2. אם כעת נוצרו ברמה 2 שלושה אלמנטים ברצף, נעלה את האמצעי ביניהם לרמה 3, וכך הלאה. אם יהיו שלושה איברים ברצף ברמה העליונה, ניצור רמה חדשה, ונשים בה את האמצעי ביניהם.
הערה: בניגוד לרשימת דילוגים רנדומלית, המפתח שמוכנס אינו בהכרח זה שמשוכפל ברמה מעל.
הוצאה - מחיקת איברים נעשית בדרך אנלוגית לגמרי לפעולת המחיקה בעץ 3-2.

<u>סיבוכיות הפעולות</u>

חיפוש - פעולת חיפוש נעשית ב- $O(\log n)$.

הכנסה - ניתן לראות כי בפעולת הכנסה, לכל היותר $\lfloor \log(n+2) \rfloor - 1$ אלמנטים גדלים לרמה נוספת.
סיבוכיות הפעולה תלוייה במימוש רשימת הדילוגים.
סיבוכיות פעולה זו תלויה בצורה בה אנו ממשים את רשימת הדילוגים.
מימוש מקובל הוא שלכל מפתח מתאים מערך של מצביעים, המציין את הצמתים היוצאים ממנו אל צמתים אחרים.
במקרה שאנו מגדילים את הרמה של המפתח, מוקצה מערך בגודל גדול יותר, ומערך המצביעים המקורי משוכפל אליו.
אם רשימת הדילוגים ממומשת כך, אזי פעולת ההכנסה תיקח $\Theta(\log^2 n)$, מכיוון שייתכן כי עבור כל הכנסה נאלץ להעתיק מערך.
הוצאה - באופן זהה להכנסה, פעולת מחיקה עלולה לקחת $\Theta(\log^2 n)$.

<u>סיבוכיות מקום (ללא הוכחה)</u>

מספר הצמתים ברשימת דילוגים דטרמינסטית 2-1 חסום על ידי $O(n)$.

<u>שיפור הסיבוכיות של האלגוריתם</u>

על ידי מימוש רשימת דילוגים כשורות של רשימות מקושרת, אנו מסוגלים להגדיל או להקטין את רמתו של צומת בזמן $O(1)$, ועל ידי כך לשפר את סיבוכיות הזמן של פעולות ההכנסה וההוצאה להיות $O(\log n)$. עם זאת, מימוש כזה מכפיל את כמות הזיכרון הנדרשת עבור המצביעים של המבנה.

מימוש של רשימת דילוגים 2-1

נציג בדפים הבאים מימוש של רשימת דילוגים 2-1 בשפת C++.

כל איבר ברשימת הדילוגים הוא מסוג CSkipNode.
הגדרת ומימוש CSkipNode להלן:

CSkipNode.h

```
#ifndef _CSKIPNODE_H
#define _CSKIPNODE_H

#include <iostream.h> // for NULL

// Skip-List Node
class CSkipNode
{
public:

      // Consturctors
      CSkipNode(int key)
            : iKey(key), SkipNext(NULL), SkipDown(NULL) { }
      CSkipNode(int key, CSkipNode* vNext, CSkipNode* vDown)
            : iKey(key), SkipNext(vNext), SkipDown(vDown) { }

      // Virtual Desturctor, to support inheritance
      virtual ~CSkipNode() {}

      ////////////////////////////////////////
      //        Skip List Functions         //
      ////////////////////////////////////////

      // Get the next node in the skip list
      CSkipNode *GetSkipNext() const { return SkipNext; };

      // Get the lower node in the skip list
      CSkipNode *GetSkipDown() const { return SkipDown; };

      // Set the next node in the skip list
      void SetSkipNext(CSkipNode *next) { SkipNext = next; };

      // Set the lower node in the skip list
      void SetSkipDown(CSkipNode *down) { SkipDown = down; };


      int GetKey() const { return iKey; };

private:
      CSkipNode *SkipNext, *SkipDown;

      int iKey;
};
```

```
inline bool operator ==(const CSkipNode& test1, const
CSkipNode& test2)
{
	return test1.GetKey() == test2.GetKey();;
}

inline bool operator >(const CSkipNode& test1, const
CSkipNode& test2)
{
	return test1.GetKey() > test2.GetKey();
}

inline bool operator <(const CSkipNode& test1, const
CSkipNode& test2)
{
	return !(operator >(test1, test2));
}


#endif
```

רשימת הדילוגים עצמה תמומש במחלקה CSkipList.
המחלקה תומכת ב-Insert ו-Search בלבד. ניתן לממש גם את Remove, על ידי שימוש באלגוריתמים של עץ 2-3, אולם פעולה זו מסובכת יחסית.

CSkipList.h

```
#ifndef _CSKIPLIST_H
#define _CSKIPLIST_H

#include "general.h"
#include "CSkipNode.h"

#include <iostream.h>

class CSkipList
{
public:

	//////////////////////////////////////
	//	SkipList Standard Functions	//
	//////////////////////////////////////

	// Consturctor.
	// Action: Initalize the Skip List.
	CSkipList();

	// Desturctor.
	// Action: Free all the elements in the skip list
	~CSkipList();
```

```
	// Insert
	// Insert new node to the skip list. The given pointer is
	// saved in the list. It doesn't create new copy of it.
	// We assume that the id doesn't exists in the skiplist
	// Gets: The node to insert
	// Returns: Error code
	Result Insert(CSkipNode *node);


	// Search
	// Search for node with given id.
	// Gets: id
	// Returns: pointer to the node if exists, else NULL.
	CSkipNode* Search(int id);


private:
	CSkipNode *Header, *Tail;


	// Insert1
	// Insert new node to the skip list. The given pointer is
	// saved in the list. It doesn't create new copy of it.
	// Gets: Node to start working on, the node to insert
	// Returns: true if a node was added on the last level
	bool Insert1(CSkipNode* Current, CSkipNode *node);

};

#endif
```

מימוש המחלקה:

CSkipList.cpp

```
#include "CSkipList.h"
#include "CSkipNode.h"

#include <iostream.h> // for NULL
#include <limits.h>
#include <stdlib.h>

// Consturctor.
// Action: Initalize the Skip List.
CSkipList::CSkipList()
{
	try
	{
		// Create the header node
		Tail = new CSkipNode(INT_MAX, NULL, NULL);
```

```
            Header = new CSkipNode(INT MIN, Tail, NULL);
            Tail->SetSkipDown(Tail);
            Tail->SetSkipNext(Tail);
      }
      catch(...)
      {
            // Handle memory problems
            cerr << "Error: Not enough memory" << endl;
            exit(1);
      }
}

// Desturctor.
// Action: Free all the elements in the skip list
CSkipList::~CSkipList()
{
      CSkipNode* First = Header;

      while (First != NULL)
      {
            CSkipNode* next = First;

            // save the next line
            First = First->GetSkipDown();

            // free all the current line
            while (next != Tail)
            {
                  CSkipNode* temp = next->GetSkipNext();
                  delete next;
                  next = temp;
            }
      }

      delete Tail;
}

// Insert
// Insert new node to the skip list. The given pointer is
saved in the
// list. It doesn't create new copy of it.
// We assume that the id doesn't exists in the skiplist
// Gets: The node to insert
// Returns: Error code
Result CSkipList::Insert(CSkipNode *node)
{
      try
      {
            if (Insert1(Header, node) && Header->GetSkipNext()-
>GetSkipNext()->GetSkipNext() != Tail)
            {
                  CSkipNode* MidNode = Header->GetSkipNext()-
>GetSkipNext();
                  CSkipNode* NewNode = new CSkipNode(MidNode-
>GetKey(), Tail, MidNode);
```

```
                  CSkipNode* NewHeader = new CSkipNode(INT_MIN,
NewNode, Header);
                  Header = NewHeader;
            }
            return SUCCESS;
      }
      catch(...)
      {
            // Handle memory problems
            cerr << "Error: Not enough memory" << endl;
            exit(1);
            return FAILURE;
      }

}

// Insert1
// Insert new node to the skip list. The given pointer is
saved in the
// list. It doesn't create new copy of it.
// Gets: The node to insert
// Returns: true if a node was added on the last level
bool CSkipList::Insert1(CSkipNode* Current, CSkipNode *node)
{
      for (; Current->GetSkipNext()->GetKey() <= node-
>GetKey(); Current = Current->GetSkipNext());

      if (Current->GetSkipDown() == NULL)
      {
            node->SetSkipNext(Current->GetSkipNext());
            Current->SetSkipNext(node);
            return true;
      }

      if (!Insert1(Current->GetSkipDown(), node)) return false;

      CSkipNode* temp;
      temp = Current->GetSkipDown()->GetSkipNext()-
>GetSkipNext();
      if (temp->GetSkipNext()->GetKey() < Current-
>GetSkipNext()->GetKey())
      {
            CSkipNode* New2 = new CSkipNode(temp->GetKey(),
Current->GetSkipNext(), temp);
            Current->SetSkipNext(New2);
            return true;
      }

      return false;
}

// Search
// Search for node with given id.
// Gets: id
// Returns: pointer to the node if exists, else NULL.
```

```
CSkipNode* CSkipList::Search(int key)
{
      CSkipNode *scan = Header;
      while (scan != NULL && scan != Tail)
      {
            while (scan->GetSkipNext()->GetKey() <= key) scan =
scan->GetSkipNext();
            if (scan->GetKey() == key && scan->GetSkipDown() ==
NULL) return scan;
            scan = scan->GetSkipDown();
      }
      return NULL;
}
```

רשימת דילוגים דטרמינסטית 1-3

הגדרה

רשימת דילוגים L תיקרא **רשימת דילוגים דטרמינסטית 1-3** אם מתקיים:
1. L היא רשימת דילוגים.
2. המרווח בין כל שני אלמנטים ב-L הוא 1, 2 או 3.

רשימה כזו ניתן להמיר באופן חד-חד ערכי לעץ 4-3-2.
עבור עצי 4-3-2 קיים אלגוריתם שבעזרתו אנו מסוגלים לבצע את ההכנסה ואת התיקונים הדרושים תוך כדי שאנו יורדים מטה במסלול החיפוש, וכך נחסך מאיתנו הצורך לנהל מחסנית של מסלול החיפוש.
נאמץ גישה זו בפעולת ההכנסה לרשימת דילוגים 1-3: בעץ פעולת הכנסה, נפצל כל מרווח של 3 לשני מרווחים של 1. בדרך זו נבטיח שהמבנה שומר על שמורת רשימת הדילוגים הדטרמינסטית, עם וגם ללא הצומת החדש שנוסיף.
באופן מפורט יותר: נתחיל את פעולת ההכנסה בראש הרשימה. אם המרווח של הצומת ממנה אנו רוצים לרדת הוא 1 או 2, אנו פשוט יורדים. אם המרווח הוא 3, ראשית נשכפל את האיבר האמצעי אל הרמה הנוכחית, תוך כדי שאנו יוצרים על ידי כך שני מרווחים בגודל 1, ואז נרד.

פעולת המחיקה נעשית באופן דומה: נתחיל את חיפוש האיבר בראש רשימת הדילוגים. לפני שנרד נרצה שכל מרווח במסלול החיפוש יהיה מרווח חוקי, אולם גדול מהמינימום, כך שאם נמחק איבר מאותה רמה, היא תשאר במצב חוקי. נעשה זאת על ידי איחוד צומת עם השכן, או על ידי שאילה מהשכן.
אלגוריתם: נתחיל את החיפוש בראש הרשימה. אם המרווח של הצומת בה אנו הולכים לרדת הוא 2 או 3, אנו פשוט יורדים. אם המרווח הוא 1, נמשיך בצורה הבאה: נסמן את המרווח ב-G. אם G הוא לא המרווח האחרון ברמה הנוכחית, אז אם המרווח שאחריו, G', הוא בגודל 1, אנו מאחדים את G ואת G'. (על ידי הנמכת האלמנט המפריד ביניהם). אם G' הוא בגודל 2 או 3, אנו משאילים ממנו צומת (על ידי הנמכת האלמנט המפריד בין G ל- G' והגבהת האיבר הראשון של G'). אם G הוא המרווח האחרון, אנו מאחדים אותו, או מלווים, מהמרווח שלפניו.
אנו ממשיכים בדרך זו עד שאנו מגיעים לרמה התחתונה, ממנה אנו מוחקים את האיבר.
מכיוון שהאלגוריתם שלנו לא מאפשר מרווחים באורך 1 במסלול החיפוש, אנו נשארים עם רשימת דילוגים חוקית.
עם זאת, ישנו יוצא דופן, והוא שאם אנו מוחקים את האיבר, עלול להיווצר מרווח יותר גדול מ-3.
בעיה זו איננה קיימת עבור עץ 4-3-2. לפיכך - כאשר אנו ממשים רשימת דילוגים זו, לא נוכל להימנע משמירת מסלול החיפוש במחסנית. לאחר מחיקת האיבר, נעבור על מסלול החיפוש מלמטה למעלה. אם המרווח גדול מ-3, (כלומר הוא 4, 5, או 6), נרים את אחד האיברים על מנת להפוך את המרווח למרווח חוקי.

אלגוריתמים אלו עומדים בסיבוכיות זמן של $O(\log n)$. אלגוריתם ההכנסה פשוט יותר מזה של רשימת דילוגים 1-2, ואילו אלגוריתם ההוצאה מסובך יותר, עקב מקרי הקצה הרבים הקיימים.

EOF